ABSTRACT

Title of Document: USING HARDWARE MONITORS TO AUTOMATICALLY IMPROVE MEMORY PERFORMANCE

Mustafa M. Tikir, Doctor of Philosophy, 2005

Directed By: Professor Jeffrey K. Hollingsworth, Department of Computer Science

In this thesis, we propose and evaluate several techniques to dynamically increase the memory access locality of scientific and Java server applications running on cache-coherent non-uniform memory access(cc-NUMA) servers. We first introduce a user-level online page migration scheme where applications are profiled using hardware monitors to determine the preferred locations of the memory pages. The pages are then migrated to memory units via system calls. In our approach, both profiling and page migrations are conducted online while the application runs. We also investigate the use of several potential sources of profiles gathered from hardware monitors in dynamic page migration and compare their effectiveness to using profiles from centralized hardware monitors. In particular, we evaluate using profiles from on-chip CPU monitors, valid TLB content and a hypothetical hardware feature.

We also introduce a set of techniques to both measure and optimize the memory access locality in Java server applications running on cc-NUMA servers. In particular, we propose the use of several NUMA-aware Java heap layouts for initial object allocation and use of dynamic object migration during garbage collection to

move objects local to the processors accessing them most. To evaluate these techniques, we also introduce a new hybrid simulation approach to simulate memory behavior of parallel applications based on gathering a partial trace of memory accesses from hardware monitors during an actual run of an application and extrapolating it to a representative full trace.

Our dynamic page migration approach achieved reductions up to 90% in the number of non-local accesses, which resulted in up to a 16% performance improvement. Our results demonstrated that the combinations of inexpensive hardware monitors and a simple migration policy can be effectively used to improve the performance of real scientific applications. Our simulation study demonstrated that cache miss profiles gathered from on-chip hardware monitors, which are typically available in current micro-processors, can be effectively used to guide dynamic page migrations in an application. Our NUMA-aware heap layouts reduced the total number of non-local object accesses in SPECjbb2000 up to 41%, which resulted in up to a 40% reduction in the memory wait time of the workload.

USING HARDWARE MONITORS TO AUTOMATICALLY IMPROVE MEMORY PERFORMANCE

By

Mustafa M. Tikir

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2005

Advisory Committee:
Professor Jeffrey K. Hollingsworth, Chair/Advisor
Professor A. Yavuz Oruc, Dean's Representative
Professor William Pugh
Professor Alan L. Sussman
Professor Peter J. Keleher

Dedication

To my parents,

Sevim and Nazif Tikir

Acknowledgements

Firstly, I would like to extend my gratitude to my advisor, Dr. Jeffrey K. Hollingsworth. His support and guidance, and above all, his endless patience and kindness made this thesis possible.

Throughout graduate school, I had many friends without whom I can not imagine my Maryland days. I am forever indebted to my dearest friends Betul Atalay and Okan Kolak who have always been there for me through good and bad times, and listened to my complaints and frustrations without any hesitation. I am always grateful to them for their exceptional support and never giving up on me. I offer my thanks to my dear friends Nihal Bayraktar, Cagdas Dirik and Ayhan Mutlu who have been a source of emotional and practical support.

I would like to extend my sincere appreciation for many people who have helped me survive with their friendship. Thanks to Funda Ertunc, Mirat Satoglu, Burcu Karagol, Fazil Ayan, Fusun Yaman, Evren Sirin, Esin and Cemal Yilmaz, Akin Akturk, Aydin Balkan. Special thanks to my officemates Chadd Williams, Bryan Buck, Jeff Odom, Nick Rutar, Ray Chen, James Waskievitz and I-Hsin Chung.

Greatest of thanks goes to my family: To my mother and father for their endless love and support at all steps of my education. To my sisters, Emine and Dilek, for always being there for me and making my life joyful. I am forever indebted to my family for their love, trust, guidance, support and confidence in me, sharing my times of despair as well as times of joy, and for believing in me.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Large cache-coherent, shared-memory multiprocessor servers are widely used for high performance computing, large-scale applications and client-server computing since these servers provide tight coupling of processors and memory resources. Several vendors now offer such servers. The SGI Altix servers[76] scales up to 512 processors, the Sun Fire servers[14] support more than 100 processors, the IBM pSeries 68[48] scales up to 64 processors, the HP Superdome[35] scales to 128 processors, and the Compaq AlphaServer GS-series[29] scales to 32 processors. The dominant architecture for the modern shared-memory multiprocessor servers is the cache-coherent non-uniform memory access (cc-NUMA).

Although cc-NUMA architectures allow construction of large shared-memory servers, data locality is an important consideration in these servers due to non-uniform memory access times. In large cc-NUMA architectures, processors have a faster access to the memory units local to them compared to the remote memory units. For example the remote and local latencies in mid-range Sun Fire 6800 servers is around 300ns and around 225ns, respectively where the remote and local latencies in high-range Sun Fire 15K servers are around 400ns and 225ns[14]. Memory intensive applications running on cc-NUMA servers may have a significant number of non-local memory accesses, which may also degrade the execution performance of the applications running on these architectures. Therefore, it is becoming more important to both analyze and optimize the memory behavior of memory intensive applications running on these cc-NUMA servers.

As the designs of the modern systems become more complex, it is getting harder to accurately simulate the complex interactions between the applications running on these systems and the underlying hardware/software components. Therefore, most processors now include hardware support for performance monitoring such as MIPS R10000[79], Compaq Alpha[20], Itanium from Intel[37], Sun UltraSPARC[68]. Similarly, shared-memory multiprocessor servers provide increasing hardware support for performance monitoring of the system interconnect such as Sun Fire Link Hardware[51]. In these complex systems, additional low-overhead hardware support for performance monitoring gives unique and valuable information about the dynamic behavior of the applications running on these systems. Even though the information gathered from these hardware performance monitors is partial or incomplete, it is still an important source of profiling information and it can potentially be used to tune the application and system behavior automatically. More importantly, the online nature of the information gathered by these monitors makes the application of online optimization techniques possible, which are particularly important for applications with behavior that dynamically changes throughout their execution.

The thesis of this dissertation is that programs can be automatically tuned at runtime using online profiling data gathered from hardware performance monitors. To validate this thesis, we introduce techniques to dynamically increase the locality of memory accesses in scientific and Java server applications running on cc-NUMA servers by placing memory pages and heap allocated objects at their preferred memory locations identified at runtime using hardware performance monitors.

The Java Programming Language[31] is gaining popularity for developing applications on a variety of platforms ranging from multiprocessor servers to embedded systems[58]. Moreover, in recent years, increasing interest is being shown in the use of Java Programming Language for client-server computing. In the client-server computing model, client applications make requests to server applications where server applications fulfill the clients' requests and return the information back to the client applications. Hence, server applications typically have large requirements for memory, bandwidth, processing power, or a combination of these. Due to the large requirements and multithreaded nature of the server applications, these applications are gaining importance as a new class of workloads for commercial shared-memory multiprocessor servers.

Java programs tend to make extensive use of heap-allocated memory and typically have significant pointer chasing, which puts pressure on the memory subsystem[58]. It has been commonly accepted that commercially important server side workloads such as the SPECjAppServer2001[63] need to run with a heap size of 1.0GB to 3.5GB to achieve high throughput[38]. Similarly, the SPECjbb2000[64] benchmark is often executed with a heap of up to 3.8GB[66]. As the disparity between processor and memory speeds continue to grow, it becomes more important to optimize the memory behavior of memory intensive applications.

This thesis describes a set of techniques for using online profiling information gathered from hardware performance monitors to automatically reduce the number of non-local memory accesses in multithreaded applications running on cc-NUMA servers. In these techniques, profiling information gathered from hardware monitors

is used to identify the preferred memory locations of memory pages and program objects such that the number of non-local memory accesses in the applications will be reduced when these memory objects are placed at their preferred locations.

We first introduce a user-level memory page migration scheme, namely dynamic page migration. In this page migration scheme, applications are profiled to determine the preferred locations of the memory pages in the memory units using hardware monitors. Then system calls are used to request the kernel to migrate the memory pages to the specific memory units. In this dynamic page migration scheme, both profiling and page migrations are conducted during the same run of the applications. The access frequencies of the memory pages by the processors are gathered continuously at runtime using hardware counters and the memory pages are migrated local to the processors accessing them most at fixed time intervals.

We also introduce a set of techniques to both measure and optimize the memory access locality of Java server applications running on cc-NUMA servers. These techniques exploit the capabilities of fine grained hardware performance monitors to provide data to automatic feedback directed locality optimization techniques. We propose the use of several NUMA-aware Java heap layouts for initial object allocation and use of dynamic object migration during garbage collection to move objects local to the processors accessing those most. To evaluate these techniques, we also introduce a new hybrid simulation approach to simulate memory behavior of parallel applications running on multiprocessor servers. Our approach is based on gathering a partial trace of memory accesses from hardware performance monitors during an actual run of an application and extrapolating it to a representative full trace

to drive the simulation. In addition, our approach uses information on heap allocations from the memory management library used in the underlying system. Our simulation approach is particularly suited to evaluate new software systems rather than new hardware components.

The hardware performance monitors we used to gather page access profiles for our dynamic page migration are centralized external monitors. These monitors are plug-in hardware that listens to all address transactions on the system interconnect in the cc-NUMA server. However, such monitors are not available in most of the systems. In this thesis, we also investigate the use of several other potential sources of profiles gathered from hardware monitors in dynamic page migration and compare their effectiveness to using profiles from centralized hardware monitors. In particular, we investigate the effectiveness of using cache miss profiles, Translation Lookaside Buffer (TLB) miss profiles from on-chip hardware monitors on processors, and the content of the on-chip TLB using the valid bit information in the TLB entries. Moreover, we also introduce an easily-installable hardware feature, called Address Translation Counters (ATC), which is specifically designed for dynamic page migration and compare its effectiveness with other sources of profiles. The ATC is a set of additional counters that is included in the TLB of the processors and gathers accurate information on access frequencies to the memory pages in the applications.

Contributions

The main contributions for my research presented in this thesis are:

Dynamic Page Migration

We introduced a user-level dynamic page migration approach that relies on the operating system kernel to provide the actual migration mechanism. Our approach

focuses on applications that are more likely to benefit from page migrations rather than trying to improve the overall system performance that is common in prior kernel-level approaches. Moreover, we demonstrate that the combinations of in-expensive plug-in monitors that sample interconnect transactions and a simple migration policy can be effectively used to improve the performance of real scientific applications even on systems with small remote to local memory latency ratios.

Dedicated Hardware Monitors for Dynamic Page Migration

We conducted a simulation based study where we investigated the use of several different types of hardware monitors and compared their effectiveness in terms of the reduction in the number of non-local memory accesses, number of page migrations triggered and execution times. We also designed and evaluated a new simple hardware feature, ATC, to accurately gather page access frequencies specifically for dynamic page migration. We demonstrate that cache miss profiles gathered from on-chip hardware monitors, which are typically available in current micro-processors, can be effectively used to guide dynamic page migrations in an application.

NUMA-Aware Java Heaps

We evaluated the potential of existing page-level locality optimization techniques on a Java server application and demonstrated that coarse-grain page-level optimization techniques may not be as effective in reducing the number of non-local memory accesses in Java server applications. Instead, we measured the memory behavior of server applications at the object level and introduced new object-centric optimization techniques including several new NUMA-aware Java heap layouts and dynamic object migration for Java server applications. We also demonstrate that hardware monitors can also be used to generate representative parallel workloads to

simulate memory behavior of parallel applications by gathering partial trace of memory accesses during an actual run and extrapolating it to a full trace.

Chapter 2: Related Work

Most of the prior research on optimizing the locality of memory accesses for applications running on cc-NUMA architectures has been in the area of page migration and replication. This chapter first describes some of the research on page migration and replication. The remainder of this chapter is grouped in topics including measuring the memory behavior of applications, optimizations proposed for improving the use of memory hierarchy, thread-local heap management, techniques that automatically adapt to the memory behavior and software simulation.

2.1. Page Placement, Migration and Replication

Noordergraaf and Zak[51] described a set of embedded hardware instrumentation mechanisms implemented for monitoring the system interconnect on Sun Fire servers. The instrumentation supports sophisticated programmable filtering of event counters. Their implementation results in a very small hardware footprint making it appropriate for inclusion in commodity hardware. Since the information gathered from these instrumentation mechanisms is based on sampling, the access frequencies of memory pages need to be approximated. Moreover, the information gathered from these instrumentation mechanisms only captures a subset of all memory accesses that also involve system interconnect for cache coherency.

Most prior page migration policies[8,43] have been in the context of non-cache-coherent NUMA multiprocessor systems. These kernel-level policies were based on page fault mechanisms and designed for multiprocessors with large remote to local latency ratios. Bolosky et al.[8] used memory reference traces to drive simulations of

NUMA page placement policies. LaRowe et al.[43] modified OS memory management modules to decide whether a page will be migrated on a page fault. In contrast, this thesis introduces page migration techniques for cache-coherent NUMA multiprocessor systems. Moreover, the page migration techniques in this thesis work at user level and migrate pages using the page access frequencies gathered from embedded hardware monitors.

Chandra et al.[13] investigated the effects of different OS scheduling and page migration policies for cache-coherent NUMA systems using Stanford DASH multiprocessors. Although they mainly focused on OS scheduling policies, they also investigated page migration policies based on TLB misses. Chandra et al. reported that page migration did not improve the response time for the workloads used due to overhead incurred by the operating system.

Verghese et al.[71] studied the operating system support for page migration and replication in cache-coherent NUMA multiprocessors. They introduced a decision tree to select the action to be taken on memory pages upon cache misses. The actions taken for a page include replication, migration and freeze, depending on the threshold values used in the decision tree. Using the thresholds that gave the best results they evaluated their approach using a simulator for SGI Origin2000 multiprocessors and workload traces of cache misses in the applications. The multiprocessor system they used also had large remote to local memory latency ratios. They reported that dynamic page placements did not yield performance gains due to overhead introduced by the operating system.

Kernel-level dynamic page placement schemes are also extensively studied in the Sun WildFire systems[11,32,50]. The Sun WildFire system is a prototype cache-coherent NUMA architecture that is built from small number of large SMP nodes and has large remote to local latency ratios. Hagersten and Koster[32] evaluated the impact of coherent page replication and hierarchical affinity scheduling on TPC-C execution. They used excess-remote-cache-miss counts to guide page placements. Noordergraaf and Pas[50] also evaluated page migration and replication using a simple HPC application. To identify memory pages for migration, they used excess misses that indicate conflict and capacity misses in a local node's cache. They reported that using a replication-only policy yielded much better performance than policies that included migration.

More recently, Bull and Johnson[11] studied the interactions between data distribution, migration and replication for the OpenMP applications. Although they particularly focused on a data distribution extension for OpenMP, they also studied the impact of page migration and replication. Their study also showed that page replication is more beneficial than migration. In contrast, this thesis introduces a user-level page migration approach for cc-NUMA servers with small remote to local memory latency (1.34:1). Moreover, our page migration approach focuses on applications that are more likely to benefit from page migrations rather than trying to increase the overall system performance.

Most recent work[75] used dynamic page placements to improve the locality for TPC-C in cc-NUMA servers. Wilson and Aglietti[75] used Verghese's dynamic page placement algorithm to tune TPC-C execution on Sybase. They used a one-second

trace of TPC-C execution and a simulator for a 4-node multiprocessor system to study the performance, bandwidth and locality of TPC-C. They used hand-tuned threshold values for dynamic page placements. Wilson and Aglietti showed that dynamic page placement could be effective if operating system overhead is hidden within the idle CPU cycles.

2.2. Analyzing the Memory Behavior of Applications

Since Java application servers, commercial database applications and Web servers are rapidly gaining importance as a new class of workloads for commercial multiprocessor servers, the memory behavior of these applications has been extensively studied. Most of these studies focused on the performance of the underlying memory subsystem or the processor. However, the techniques described in this thesis gather per-thread memory behavior information of applications in terms of memory pages and heap allocated objects.

Karlsson et al.[39] presented detailed memory system behavior of the application servers in ECperf[25] and SPECjbb2000[64] benchmarks running on commercial cc-NUMA multiprocessor servers. They used hardware counters in SunE6000 and SunE15K and conducted uniprocessor simulation to investigate the impact of different memory subsystem parameters. They found that a large fraction of the working data sets is shared among processors. In another study, Karlsson et al.[38] focused on the memory behavior of ECperf benchmark and found that many of the L2 cache misses in ECperf are satisfied by the caches of the neighbor processors.

Marden et al.[47] studied the memory system behavior of the server applications in the Perl and Java versions of SPECweb99 benchmark on a 4-node multiprocessor

server. Using simulation, they measured the cache behavior of the server applications in both implementations. They reported that the cache miss rate becomes worse for the Java implementation when the size of the cache is increased due to likelihood of shared data residing in remote caches.

Chow et al.[19] presented uniprocessor performance characteristics of transactions from the ECperf benchmark. They used hardware counters to measure the program behavior and presented correlations between both the mix of transaction types and the system configurations in terms of different performance characteristics including L2 cache misses. Similarly, Luo and John[45] also studied the characterization of multithreaded Java server workloads. They compared the VolcanoMark and SPECjbb2000 Java server benchmarks with SPECint2000[60] benchmarks in terms of processor performance. Luo and John mainly investigated the effects of multithreading and measured the number of cache misses in the multithreaded Java server applications.

Barroso et al.[2] studied the memory behavior of commercial workloads including online transaction processing (OLTP) on a multiprocessor server. They used both hardware counters and a detailed software simulation. They reported that OLTP workloads are particularly sensitive to memory and cache-to-cache data transfer latencies, especially in the presence of large L2 caches. Similarly, Ranganathan et al.[54] studied the database workloads using detailed simulations. They also reported that a large fraction of the communication misses in OLTP exhibit migratory behavior by accesses to a small faction of migratory data. Ailamaki et al.[1] examined four commercial DBMSs running on an Intel Xeon processor. Using hardware counters,

they measured the execution time breakdown of these applications into different system components. They reported that 90% of the memory stalls are due to L2 data cache and L1 instruction cache misses.

The memory behavior of SPECjvm98[61] benchmarks has also been studied. In these studies, information gathered has been presented in terms of different characteristics. Shuf et al.[58] analyzed the memory behavior of SPECjvm98 benchmarks and pBOB[3]. Using both hardware counters and simulation using heap access traces, they gathered information on the distribution of heap accesses and TLB and cache misses over objects, arrays, and virtual method tables. Shuf et al. reported that the number of hot fields in applications is very small, and that most of data TLB and cache misses are due to the frequently accessed objects.

Kim and Hsu[40] studied the memory behavior of SPECjvm98 benchmarks using an exception based tracing tool. They investigated the lifetime characteristics of the objects, the temporal locality and the impact of cache associativity on cache miss rates. Kim and Hsu reported that the overall cache miss ratio is increased due to garbage collection, which suffers higher cache misses compared to the application itself. They also observed that Java programs generate substantial number of short-lived objects but the size of frequently accessed long-lived objects is more important to the cache performance.

Yang et al.[77] measured the allocation latencies, garbage collection elapsed time and the object lifetime in the SPECjvm98 benchmarks. They triggered garbage collection at every 50K memory allocations and at fixed time intervals and compared the lifespan of the objects in the SPECjvm98 applications measured using both

approaches. Yang et al. showed that time based approach yields fewer garbage collections while maintaining the heap residency same as the space based approach.

2.3. Optimizations to Create and Preserve Locality

The prior research on pointer intensive applications with dynamic memory allocation mostly studied techniques to create and preserve locality of the applications through better use of caches. Due to difficulty of making good decisions at compilation time, these techniques mostly used profiling information gathered during separate runs. In contrast, the goal of this thesis is to both gather online profiling information from hardware monitors and to tune applications during a single run.

Shuf et al.[57] presented an allocation-time object placement technique that co-locates heap objects based on the notion of prolific types. They also introduced a garbage collection time graph-traversal technique that tries to improve the locality of the garbage collection and surviving objects by first traversing the objects that reside close to each other before traversing their children. Shuf et al. reported that their techniques significantly improve application and garbage collection performance.

Yardimci and Kaeli[78] presented profile-guided techniques to allocate heap objects in a cache-conscious way for C applications. Using cache miss information, they modified malloc allocation routine to place heap objects in appropriate regions to reduce conflict misses in L1 data caches. They also divided the memory allocations into phases and built a temporal relationship graph (TRG) for each allocation phase. Using the TRGs, Yardimci and Kaeli allocated contemporaneously accessed objects into neighboring regions in the heap and data cache. In both techniques, they used stack pointer content and the object size as predictors of the allocation phases.

Calder et al.[12] presented a compiler-directed technique for cache conscious data placement for pointer-intensive applications. In their approach, applications are first profiled to characterize how data is used and then using the profiles, address placements of call stack, global constants, global variables and heap objects are decided to reduce the conflict misses. They guided the placements of the objects by building a TRG among all objects. Calder et al. showed that cache-conscious placement produces significant gains for global variables and call stack, but modest gains for heap objects.

Chilimbi et al.[15-17] described several techniques for improving cache locality of pointer-intensive applications. Chilimbi et al.[15] described two techniques, structure splitting and field reorganization, for cache conscious structure definition for C and Java applications. In their technique, Java objects are split into a hot and a cold portion using previously gathered field access frequencies. They also implemented a tool that produces structure field reordering recommendations to the programmer using structure field affinity graphs. Similarly, Chilimbi et al.[16] demonstrated that good data organization and layout can improve the spatial and temporal locality of the pointer-intensive C applications. They described a data organizer tool for tree like structures to which the programmer supplies a function to traverse the structures for allocation. They also implemented a cache-conscious heap allocator that uses programmer-supplied hints and attempts to co-locate contemporaneously accessed data on the same cache line. In another study, Chilimbi et al.[17] described how a copying garbage collector can be used for cache-conscious object layout using real-time profiling information for Cecil applications. They

introduced a generational garbage collector that reorganizes data for a cache-conscious layout, in which long-lived objects with high temporal affinity are placed next to each other. They constructed object affinity graphs for each generation and used these graphs to traverse live objects in the heap during garbage collection intervals. Chilimbi et al. showed that significant reduction in cache misses is possible through the techniques described in [15-17].

Truong et al.[69] presented profile-guided techniques for structure field reorganization and instance interleaving in the pointer intensive C applications. They used source code modifications in the structure definitions. For field reorganization, Truong et al. grouped fields of each data structure that are often referenced together to fit in the same cache line. Kistler and Franz[41] also presented a profile-guided technique that uses temporal profiling information for structure field reordering. Their technique tried to maximize the spatial locality of individual data members by assigning fields that are accessed close in time to the same cache line. To identify the fields that are accessed closely in time, they constructed TRGs using path profiles.

Seidel and Zorn[56] investigated profile-driven techniques for allocation intensive programs to predict object references and lifetimes using variety of information available at the time of object allocation. Later, they used profiles to reduce the number of page faults and to increase the spatial locality by segregating objects into different heap areas using the objects' access frequencies and lifetimes. They also evaluated the impact of different predictors such as stack pointer, path pointer and contents of the call stack at object allocation times and showed that references to the heap objects are highly predictable.

Ding and Kennedy[23] presented a dynamic approach for improving both computation and data locality in irregular programs. They introduced locality grouping that reorders data accesses to improve spatial use of the objects and dynamic data packing that places data accessed at close intervals into the same cache line. They investigated effectiveness of different packing techniques including first-touch packing that classify objects in the order they appear, group packing that classify objects according to their reuse patterns, and a hybrid technique. Ding and Kennedy also proposed that compilers could provide support to control the overhead of dynamic data packing within acceptable limits.

2.4. Escape Analysis and Thread-Local Heaps

Escape analysis is a static whole-program analysis technique that determines whether the lifetime of data exceeds its static scope such as the scope of a method and/or a thread. Escape analysis has been used for optimization of stack allocation and thread synchronization elimination. Unlike escape analysis, the techniques in this thesis work on objects shared among threads and require dynamic analysis of object access frequencies during program executions.

Gay and Steensgaard[28] presented an escape analysis algorithm for Java applications and used the analysis results for stack allocation of the objects. They considered an object to have escaped from the scope of a method if a reference to the object is returned from the method or if a reference to the object is assigned to a field of another object. They reported that 10-20% of the objects are local to the methods and can be allocated on stack.

Choi et al.[18] and Blanchet[7] used escape analysis for stack allocation and elimination of thread synchronizations. Choi et al.[18] used connection graphs to capture the connectivity relationship among object references and performed flow-sensitive reachability analysis on the connection graphs to determine objects local to methods and threads. Blanchet[7] used integer height types to encode object reference relationship and sub-typing. He used a two-phase flow-insensitive analysis to identify objects local to methods and threads. Both Choi et al. and Blanchet reported that escape analysis is effective to determine significant number of objects that do not escape their local scopes.

Salcianu and Rinard[55] presented a combined pointer and escape analysis for multithreaded applications that use region based memory allocation. Their algorithm constructs parallel interaction graphs that store reference relationship among objects accessed by multiple threads and capture the objects that do not escape a given multithreaded computation. They used fixed-point computation to obtain a single parallel interaction graph that reflects the interactions between all parallel threads. Salcianu and Rinard used the analysis results to verify correct use of region-based allocation, eliminate dynamic checks associated with the use of regions and eliminate unnecessary synchronizations.

Memory management techniques using thread and processor local heaps have also been studied for multithreaded applications. These techniques are mainly designed to eliminate memory management synchronizations among multiple threads. Even though these techniques relate to the memory management policy described in this thesis, we allocate objects local to the processors that access those most.

Berger et al.[5] introduced a fast, highly scalable memory allocator, Hoard, for multithreaded applications that avoids false sharing and is memory efficient. Hoard is designed to enable multithreaded applications to achieve scalable performance on shared-memory multiprocessor servers. Hoard maintains a global heap and many per-processor heaps to allocate objects in the multithreaded applications. When a per-processor heap usage drops below a certain fraction, Hoard transfers a large fixed-size chunk from a per-processor heap to the global heap to be used by other processors. Berger et al. showed that Hoard keeps memory blowup and synchronization costs to a constant factor.

Steensgaard[65] and Domani et al.[24] investigated the benefits of using thread-local heaps to improve garbage collection performance in Java applications. Steensgaard proposed an automatic memory manager to allocate a section of the heap for each thread where the thread allocates its local objects, and to allocate another section of the heap for objects shared among threads. He used thread escape analysis to identify thread-local objects. Similarly, Domani et al. assigned each application thread a partition of the heap in which the thread allocates its local objects. Domani et al. dynamically monitored applications to determine local objects and marked all descendents of a local object as global when the object becomes descendent of a global object. When a thread requires space, both Steensgaard and Domani et al. first triggered minor garbage collection to collect thread-local heap spaces.

2.5. Automatically Adapting to the Memory System Behavior

Bennet et al.[4] described an adaptive memory coherence mechanism for distributed shared memory architectures in which several different mechanisms, each appropriate

for a different access pattern of shared objects, are used. They showed that large percentage of all accesses to shared objects could be characterized by a small number of categories of access patterns. For each access pattern, Bennet et al. developed an efficient coherency mechanism and described methods to identify objects that exhibit the pattern. They compared write-invalidate and write-update coherence policies with their adaptive policy and showed that adaptive cache coherency outperformed the standard mechanisms.

Cox and Fowler[21] described a cache-coherency protocol that automatically classifies cache memory blocks as migratory or shared and switches between a sub-protocol optimized for migratory data and other appropriate for shared data. They identified a data block as migratory, if there are two cached copies of the block and the processor that requested the invalidation is different from the processor that previously requested the invalidation at the time of a write-hit to the shared block. Cox and Fowler used replicate-on-read miss policy for shared data blocks and migrate-on-read-miss policy for migratory data blocks.

Bershad et al.[6] described an adaptive technique for reducing the number of conflict misses in physically indexed direct-mapped caches using online information gathered from an inexpensive hardware feature, Cache Miss Lookaside (CML) buffer. The CML buffer records and summarizes a history of recent cache misses in terms of list of pages and the number of cache misses on pages. CML information is used by the operating system to dynamically adjust the virtual to physical address mappings to reduce the number of conflict-induced cache misses. When the number of cache

misses occurred in a page exceeds a programmable threshold, operating system relocates all but one of the conflicting pages to prevent potential cache conflicts.

Horowitz et al.[34] proposed a hardware feature, informing memory operations, as a means of both dynamically measuring the memory behavior and adapting to the measured behavior. An informing memory operation allows an application to gather information on whether a memory access hits or misses the caches. Horowitz et al. described two forms of informing memory operations; The first form used a combination of cache outcome condition code and a special branch-and-link instruction to enable transfer of execution to a code segment when a cache miss occurs. The second form used low-overhead traps. Horowitz et al. also proposed possible uses of informing memory operations for performance monitoring, software-controlled prefetching and multithreading, and enforcing cache coherency.

Glass and Cao[30] described a virtual page placement policy, SEQ, based on the observed pattern of page faults. SEQ normally performs LRU replacement. It also monitors page faults as they occur and detects long sequences of faults to contiguous virtual addresses. When such sequences are identified, SEQ switches to a pseudo MRU replacement on the sequences. In SEQ, the replacement page is chosen from a sequence that faulted most recently and of which the length is greater than a programmable threshold. If no such sequence exists, SEQ falls back to LRU replacement policy. Glass and Cao showed that SEQ performs significantly better compared to LRU for the applications that exhibit sequential access patterns.

2.6. Software Simulation

Execution-driven simulators, such as MINT[70], Augmint[49], RSIM[53] and Simics[46], are widely used to gather information on the memory behavior of applications for a desired memory architecture. These simulators are generally concerned with timing characteristics of the program components or the memory subsystem. Even though traces for memory accesses in an application can be gathered with these simulators, due to difficulty of accurately simulating complex interactions between systems and applications running, profiles gathered from these simulators are not as representative as the ones gathered from hardware performance monitors.

Trace-driven simulators combined with trace generation tools, such as Dinero IV[26], MPTrace[27] and QPT[44], are used to simulate the interactions between the underlying memory subsystem components and applications. However, these simulators are generally slow due to the fact that they simulate full caches using traces gathered. Moreover, static instrumentation used to gather traces may have a significant impact on the trace quality due to perturbation to the memory behavior of the application under test. In our techniques, we eliminate potential perturbation by gathering information on objects that survive at least one garbage collection and by sampling memory accesses via low-overhead performance monitors.

The Java Virtual Machine Profiler Interface (JVMPI)[74] defines a general purpose mechanism to obtain profiles from a Java VM. However, the JVMPI provides information about heap allocations using JNI handles and does not provide information about where objects are stored in terms of memory addresses.

DeRose et al.[22] introduced a data collection framework and family of cache analysis tools, SIGMA. The SIGMA environment provides detailed cache information by gathering memory reference data using software-based instrumentation. The tool can also assist in perturbation analysis to determine performance variations caused by changes to system hardware or software. The MetaSim[59] tracer generates detailed information about load/store unit usage for an application and captures dynamic memory address information during an instrumented run under a desired memory subsystem. The address stream is later processed to find the memory access patterns.

Kurc et al.[42] presented a simulation-based framework for data intensive parallel applications on parallel machines. They used application emulators that provide parameterized models of applications to be able to scale applications in a controlled way and suite of simulators for large number of processors. An application emulator is a simplified version of the real application with necessary communication and computation characteristics and it provides a specification of the behavior of an application to a simulator. In comparison, our technique is more automatic in workload generation and uses profiles gathered from hardware monitors.

Chapter 3: Dynamic Page Migration

In this chapter, we introduce a dynamic page migration scheme that profiles applications to determine the preferred location for each memory page using hardware monitors. We then use system calls to request that the kernel move memory pages to their preferred locations.

In our dynamic page migration algorithm, both profiling and page migration are conducted during application execution. For each memory page in the application, we continuously gather data from hardware performance monitors about the processor most frequently accessing the page. At fixed time intervals during the application's execution, pages are migrated to a memory unit that is local to the processor that most frequently accesses each page.

Although page migration has been extensively studied in prior research, our dynamic page migration approach demonstrates several novel features. First, our goal is not to introduce a new page placement policy. Instead, we demonstrate that the combinations of in-expensive plug-in hardware monitors that sample information about interconnect transactions and a simple page migration policy can be used effectively to improve the performance of real scientific applications. The plug-in hardware we used in this research is commercially available from Sun Microsystems.

Second, even on multiprocessor systems with small remote to local memory latency ratios, optimizing page placement still provides substantial benefit to some applications. The remote and local latencies in the Sun Fire 6800 servers we used in our research are approximately 300ns and 225ns respectively (i.e. a remote to local

memory latency ratio of 1.33:1). This is quite low for a NUMA system, and previous research on optimizing page placements has tended to focus on systems with much larger remote to local memory latency ratios. We believe our techniques will be more effective on systems with large remote to local memory latency ratios.

Third, using the information provided by hardware monitors that gather information based on physical addresses rather than virtual addresses is accurate enough to guide page migration and eliminates the need for getting virtual address information via hardware monitors.

3.1. Hardware and Software Components

In this section, we describe the hardware and software components used in our dynamic page migration research. We first describe the architecture of the Sun Fire servers. We next describe the Sun Fire Link hardware monitors for the Sun Fireplane system interconnect, which we used to measure memory access behavior. Finally, we give a brief explanation about the system calls that we used.

3.1.1. Sun Fire Servers

The Sun Fireplane interconnect is Sun's fourth generation of Symmetric Multiprocessor Systems (SMP) interconnect. The Sun Fireplane interconnect is implemented with up to four levels of interconnect logic depending on the number of processors in the server[14]. In medium and large-sized Sun Fire servers, processors and memory units are grouped together on system boards[68]. Each system board contains 4 processors and 4 memory units local to the processors.

In Sun Fire servers, the transfer time to move a data block from a memory unit to the requesting device is non-uniform depending on the system boards the memory unit and requesting processor are on. Processors on a system board have faster access to the memory banks on the same board (local memory) compared to the memory banks on another board (non-local memory). For example, back-to-back latency measured by a pointer-chasing benchmark in a Sun Fire 6800 server with 750MHz CPUs is around 225ns if the memory is local and 300ns if the memory is non-local. Moreover, the back-to-back latencies in larger Sun Fire 15K servers are even larger and are around 225ns if the memory is local and 400ns if the memory is non-local.

The Sun Fire 6800 server is a mid-range cc-NUMA architecture based on the UltraSPARC III processors and Sun Fireplane interconnect. It supports up to 24 processors and 24 memory units. The processors and memory units in these servers are grouped into 6 system boards. Each processor has its own on-chip and external caches. Mid-range Sun Fire systems use a single snooping coherence domain that spans all the devices connected to a single Fireplane address bus.

3.1.2. Sun Fire Link Hardware Counters and Bus Analyzer

In a cache-coherent shared-memory multiprocessor, the system interconnect is often the performance-limiting component for some applications[51]. Moreover, due to complex interactions among the processors and devices that utilize the system interconnect, it is difficult to analyze the performance of the system interconnect. Due to high transaction rates in these systems, gathering a complete set of interconnect transactions is not practical. Instead, these systems often have additional hardware monitors to count and sample the system transactions. Even though the information

collected by these hardware monitors is incomplete, it is still an important source of profiling information[51].

For our dynamic page migration scheme, we use the Sun Fire Link hardware monitors[51] to gather profiling information for page migration (Shown in Figure 1). The Sun Fire Link hardware monitor counts and samples the transactions on the address bus of the Sun Fireplane interconnect. These monitors were developed as part of a system to cluster multiple systems together, thus they listen to the address bus of the system interconnect.

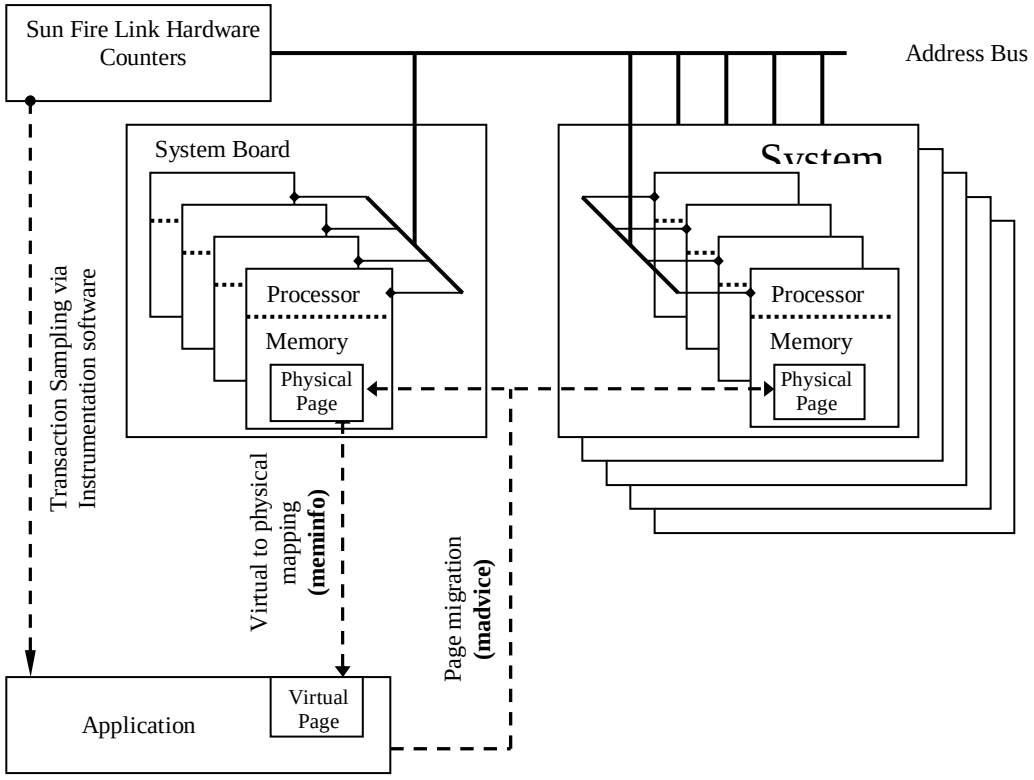


Figure 1: Framework for our dynamic page migration scheme

The Sun Fire Link Monitors consist of two 32-bit counter registers, a programmable control register that activates the counters, two registers to filter transactions based on transaction type, and two sets of mask and match registers to filter transactions based on other parameters, such as physical address range and the

device identifier. In addition to counter registers, the Sun Fire Link Bus Analyzer has an 8-deep FIFO that records a limited sequence of consecutive interconnect address transactions. Each recorded transaction includes the requested physical address, the requestor device id, and the transaction type. The bus analyzer is configured with mask and match registers to select specific address ranges, processors or transaction types.

Even though the Sun Fire Link counters and bus analyzer provide useful information about the addresses and requesting processors in the transactions, the information is at the level of physical addresses. To accurately evaluate the memory performance of an application, the address transactions have to be associated with virtual addresses used by the application. This requires us to reverse map physical addresses back to virtual addresses. We used the *meminfo* system call in Solaris 9 to create a mapping between physical and virtual memory pages in the applications.

3.1.3. System Calls in the Solaris 9 Operating System

To ensure the reusability of local caches in the processors, each application thread should be scheduled on the same processor, if possible, throughout its execution[62]. To ensure the reusability of local caches and to accurately count page access frequencies by processors independent of thread scheduling, we explicitly bind application threads to the processors in the system. We bind application threads to the processors in a round robin fashion using the *processor_bind* system call in Solaris.

Solaris places each physical memory page into the memory that is local to the first processor that touches the page. However, first-touch page placement may result in

non-local placement of a page relative to the processor that accesses it the most, which may have a significant impact on memory performance of the application.

To move pages, we use the move-on-next-touch feature of the *madvise* system call in Solaris 9. Using the move-on-next-touch feature, we request the operating system to move a range of virtual memory onto the local memory of the processor that next touches the range.

3.2. Methodology

Our dynamic page migration algorithm consists of two different modules. The first module gathers profiling information using the Sun Fire Link counters and bus analyzer. The second module migrates memory pages using the profiling information gathered by the first module. In our page migration approach, we insert instrumentation code into the application to gather profiling information, to migrate the memory pages, to bind application threads to processors and to detect the application termination.

We used Dyninst[9] to insert instrumentation code into the application being analyzed. Dyninst is a library that permits the insertion of code into a running program. The Dyninst library provides a machine independent interface to permit the creation of tools and applications that use runtime code patching.

For our dynamic page migration algorithm, instrumentation code is inserted at the entry of the *main* function, exit point(s) of *thr_create* function, and the entry of *exithandle* function. The instrumentation code that is inserted at *main* loads a shared library that creates additional helper threads for gathering profiling information and migrating memory pages. The instrumentation code inserted at the exit point(s) of

thr_create calls the *processor_bind* system call to explicitly bind the newly created application threads to available processors in a round robin fashion. The helper threads are bound to dedicated processors and the remaining processors are used to bind the other threads in the application. The instrumentation code inserted at the entry to *exithandle* detects the application termination and cleans up the hardware monitors and software libraries.

Our dynamic page migration algorithm is a two-phase algorithm. It creates two helper threads, one for profiling and another for page migration. The profiling thread samples the interconnect transactions and updates the access frequencies of the memory pages for each system board. The migration thread stops the execution of all other application threads at fixed time intervals and triggers page migration based on the profiling information gathered. In addition, to prevent memory pages ping-ponging between memory units, we freeze memory pages that have been migrated recently for a fixed number of page migration iterations (We freeze a page after 3 consecutive iterations). Thus, the memory pages are migrated at fixed time intervals and a page may be migrated more than once throughout application execution.

Our migration algorithm does not use a minimum access frequency threshold to trigger the migration of a page. At every migration interval, regardless of the number of accesses to a page, the page is considered as candidate for migration. Alternatively, we could limit migration to the pages with a minimum number of accesses or cache misses and thus migration overhead would potentially be eliminated for pages with little contribution to the application's memory time.

3.3. Preliminary Experiments

In this section we will discuss the results of our preliminary experiments to ensure that we could accurately sample interconnect transactions in the applications being analyzed, and that placing pages local to the same board as the processor can have a significant impact on the memory performance of an application.

3.3.1. Interconnect Transaction Sampling

We sample the interconnect transactions using the Sun Fire Link hardware monitors and approximate the access frequencies for the memory pages. However, for sampling to be effective, the sampling technique has to be representative of all transactions that occurred during the execution of the application being analyzed.

One approach to sample interconnect transactions using the Sun Fire Link bus analyzer is to continuously sample at the maximum speed of the interconnect instrumentation software. We refer to this sampling scheme as *maximum-rate* sampling. Maximum-rate sampling does not capture a compete set of transactions, but it tries to sample as many transactions as possible. Alternatively, transactions can be sampled at fixed time intervals or at every N_{th} transaction occurrence, where N is a constant that defines the interval of sampling[10]. In this thesis, we refer to sampling at every N_{th} transaction occurrence as *interval* sampling.

We conducted a series of experiments to compare how representative the maximum-rate and interval sampling techniques are of all transactions. To objectively compare the two sampling techniques we designed a distance metric *D* that given a set of transactions and a set of samples from the set, measures the percent difference between the values of a property for these sets. The property we used in our

experiments is the ratio of transactions requested by a specific processor to the total number of transactions. This metric indicates how much a set of transactions deviate from another set of transactions in terms of memory behavior. Thus, the closer the value of our distance metric is to 0, the more representative the set of sampled transactions is of the set of all transactions. Since the Sun Fire Link counters can accurately count the number of transactions as well as the number of transactions from a given processor, we counted both of these values and compared them with samples taken via Sun Fire Link bus analyzer to approximate the sampling error of sampling techniques.

For each experiment, we configured one of the two counters in the Sun Fire Link hardware monitors to count the number of transactions requested by a selected processor P, denoted C_P. The other counter is configured to count all transactions, C_A. Using the Sun Fire Link bus analyzer we also sampled interconnect transactions and recorded the number of transactions sampled, denoted S_A. In the set of sampled transactions, we count the number of transactions that are requested by processor P, denoted S_P. We calculate the ratios for the set of sampled transactions and the set of all transactions as $R_{Sample} = S_P/S_A$ and $R_{All} = C_P/C_A$, respectively. We define the distance as $D = ABS(R_{Sample} - R_{All}) / R_{All}$. That is, the distance metric gives an insight as to how far the set of sampled transactions deviate from the set of all transactions.

We conducted a series of experiments for a set of processors while running an OpenMP version of the CG benchmark from NAS Parallel benchmark suite[52]. For our experiments, we ran CG with 6 threads using the input set of size B. We repeated

the experiments with different sampling intervals in which samples taken at every 64, 256, 1024 and 4096 transactions.

Table 1 presents the results of the experiments conducted to compare how representative the sampled transactions are of all transactions. In Table 1, the second column gives the distance values for maximum-rate sampling. The third to sixth columns give results for interval sampling with different interval values. In Table 1, the rows that are labeled with processor identifiers give the distance between the set of all transactions and the set of sampled transactions with respect to that processor. The second from the last row averages the distance values of all experiments.

Table 1 shows that maximum-rate sampling can sample about 18% of all transactions. Table 1 also shows that for maximum-rate sampling, for each processor, the distance metric is significantly higher compared to interval sampling. Moreover, for maximum-rate sampling, the average distance over all processors is 0.56, which shows that the set of sampled transactions is quite different from the set of all transactions (Recall a value of 0 for distance *D* is perfect sampling correlation).

	Max-Rate Sampling	**Interval Sampling**			
		4K	**1K**	**256**	**64**
Processor 0	0.51	0.03	0.03	0.03	0.09
Processor 1	0.61	0.04	0.04	0.04	0.09
Processor 2	0.47	0.01	0.02	0.02	0.23
Processor 3	0.58	0.00	0.01	0.01	0.02
Processor 4	0.65	0.02	0.02	0.02	0.12
Processor 5	0.57	0.03	0.02	0.03	0.15
Average Dist.	0.56	0.02	0.02	0.02	0.11
% Sampled	17.56	0.19	0.78	3.07	9.75

Table 1: Distance values for maximum-rate sampling and interval sampling

During maximum-rate sampling, the maximum number of transactions the instrumentation software can record bounds the number of samples that can be taken for a processor. Thus, if a processor requests transactions faster than the maximum rate the instrumentation software can read, many transactions for the processor will not be recorded. Similarly, if a processor requests transactions slower than the rate of instrumentation software, almost all of its transactions will be recorded as samples. Thus, maximum-rate sampling results in a skewed distribution of sampled transactions with respect to the level of memory system activity on processors and the sample set does not accurately represent all transactions.

Table 1 also shows that for interval sampling, the distance values depend on the sampling rate. The distance values are low and similar to each other except for the experiments where transactions are sampled at every 64 transactions. In particular, if the samples are taken at every 256 transactions or more, the set of sampled transactions is fairly representative of all transactions. Table 1 also suggests that for interval sampling, if the rate that samples are taken exceeds 5% of all transactions, the set of sampled transactions becomes less representative.

To further investigate how representative the samples for larger sampling interval values, we also conducted experiments varying the sampling interval up to every-128M address transactions. In addition, for each experiment, we also recorded the number of distinct pages that are included in the set of sampled transactions. Figure 2 presents the average sampling error (left y-axis) and the percentage of distinct pages sampled (right y-axis) in the application for the intervals we tested.

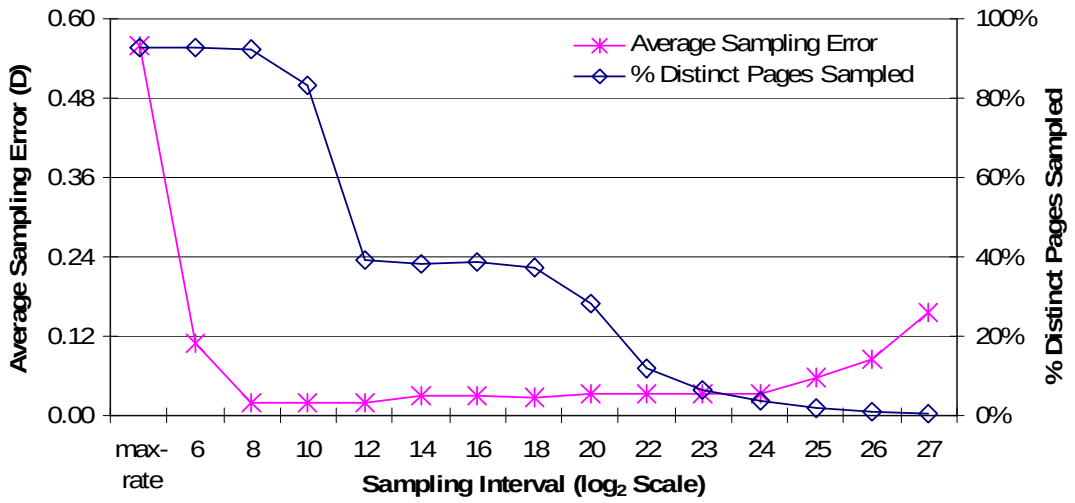


Figure 2: Average distance and percentage of pages sampled in CG (B)

Figure 2 shows that average sampling error is the highest for max-rate sampling and it starts decreasing dramatically as the sampling interval increases. Moreover, average error stays low and steady for a large range of sampling intervals starting at every-256 transactions sampling to every-8M transactions sampling. Average sampling error starts to increase again after every-16M transaction sampling due to the fact that the number of samples taken is not large enough to accurately characterize all transactions in the application.

Figure 2 also shows that for max-rate sampling, 93% of all pages in the application are included in the samples taken. Similarly, for smaller intervals, the percentage of distinct pages sampled is around 90% for interval sampling. However, as the sampling interval increases, Figure 2 shows that the percentage of distinct pages sampled in interval sampling decreases dramatically, resulting in many pages not included in the set of sampled transactions. More importantly, Figure 2 shows that even though interval sampling generates more representative samples, the percentage of the pages included in the samples decreases as the sampling interval increases.

Since transaction sampling competes for bus bandwidth with the application being measured, it is also necessary to quantify the bus load due to the sampling technique used. To quantify the bus load of each sampling technique, we conducted an experiment where we counted the number of address transactions due to accessing the hardware monitor. From this, we calculated the additional bandwidth consumed.

Our experiments showed that both maximum-rate and interval sampling produce the same bus load of around 0.5MB/sec (0.005% of the maximum data bandwidth). This is due to the fact that the dominant part of the bus load is produced by sampling the counter contents to determine whether it is time to take a sample rather than getting the sample. If the counters had an interrupt on overflow feature (common in current on-chip CPU hardware monitors), we could eliminate much of this bus load.

3.3.2. Impact of Local Page Placement

Before testing the benefits of our page migration scheme on multithreaded applications, we wanted to assess the impact of page placement on the memory performance of a single threaded application. We designed a simple application that sequentially traverses over the elements of an array repeatedly. Before each array element is accessed, the cache line containing the element is invalidated and the access is satisfied by the memory in which the array pages are placed. Note that this application is designed to exercise memory heavily and real applications would not have as many cache misses.

We conducted experiments running the single threaded application under local and non-local page placement, and we measured the total time spent to access array elements. Moreover, to eliminate factors such as pre-fetching or speculative loads, we

also implemented a variant of this benchmark that uses a random number generator to decide on the next element to be accessed.

Table 2 presents the memory access times for our test programs. In Table 2, the first column lists the applications, where each row is labeled by the pattern in which the array elements are traversed. The second and third columns give the memory access times for local and non-local placements of the array pages, respectively. The fourth column lists the slowdown ratios when array pages are placed non-locally compared to being placed locally on the processor running the application.

Array Access Pattern	Array Page Placement		Slowdown
	Local	Non-Local	
Sequential	546.6	685.8	1.25
Strided	659.7	810.1	1.23
Random	512.5	606.0	1.18

Table 2: Array access times in seconds for local and non-local page placement

For each program, Table 2 shows a significant slowdown in array access times when array pages are placed non-local to the processor running the application compared to placing array pages locally. The slowdown ratios for array access times range from 1.18 to 1.25. More importantly, Table 2 shows that the slowdown due to non-local page placement is directly proportional to the back-to-back latencies measured by the pointer-chasing latency benchmark.

We noticed a form of intra-board locality in the Sun Fire servers. That is, although the array pages are local, the choice of the processor from the group of processors on the same system board also has an impact on the array access times. Table 3 presents the array access times for each application when different processors in the same system board are used to execute the application. In each execution, array pages are

placed identically. In Table 3, the second column presents array access times when the test programs run on the first processor in a system board where the third column presents array access times when they run on the second processor.

Table 3 shows that the programs spent 7-11% more time traversing the array elements when they are bound to the second processor of the system board compared to when they are bound to the first processor even though the array pages are placed local to the processors. We believe intra-boards variations in array access times are to due to whether the array pages are placed on the memory banks controlled by the processor running the application or on the memory banks controlled by another processor in the same system board. We also believe increasing the number of memory banks controlled by each processor will reduce the intra-board variations.

Array Access Pattern	Processor on System Board		Slowdown
	CPU 0	CPU 1	
Sequential	546.6	604.3	1.11
Strided	659.7	715.4	1.08
Random	512.5	546.0	1.07

Table 3: Intra-board variation in array access times

3.4. Page Migration Experiments

To investigate the effectiveness of our dynamic page migration approach on the performance of real applications, we conducted experiments using the OpenMP C implementation of the NAS Parallel Benchmark suite[52]. We chose applications with different sizes ranging from B to C (large data set sizes) such that each application would have a similar memory footprint. We compiled the applications using Sun's native compiler, Sun C 5.5 EA2, with optimizations (-xopenmp=parallel and -O3) on to support parallelized code.

We conducted all of our experiments on a 24-processor Sun Fire 6800 with 24GB of main memory (as described in Section 3.1). The system clock frequency is 150MHz. The processors are 750MHz UltraSPARC III. The memory in each system board is 8-way interleaved where each processor controls two banks of memory. The Sun Fire Link hardware is plugged into an I/O drawer in this system. The Sun Fire Link instrumentation has full visibility into all transactions on Fireplane interconnect.

To quantify the benefits of our dynamic page migration approach, we conducted a series of experiments with and without page migration. For all applications, we measured both the original execution times and the execution times when pages are migrated using our dynamic page migration approach. For each application, we also measured the percentage reduction in the number of non-local memory accesses when memory pages are dynamically migrated compared to its original execution.

We ran all applications with 12 threads on 6 system boards of the Sun Fire 6800 server. To eliminate any possible contention due to resource sharing among processors, we scheduled two threads on each system board. We sampled interconnect transactions at every 1024 transactions.

As explained in Section 3.2, we insert instrumentation code into the application using the Dyninst library. For each application, the instrumentation overhead is a one-time overhead since the Dyninst library has a capability of saving instrumented executables for later reuse. Moreover, the instrumentation overhead for our page migration approach is independent from the execution times of the applications we analyzed. We measured the instrumentation overhead for all applications for our dynamic page migration approach and it is typically around 2 seconds.

3.4.1. Sensitivity to Migration Interval

For the experiments with page migration, the migration interval is given as a parameter to out dynamic page migration scheme. Thus, it is also important to investigate the impact of the migration interval on the effectiveness of our migration approach. To investigate the impact of migration intervals and choose the migration interval for the experiments, we conducted a sensitivity analysis in which we ran each application under different migration intervals ranging from 1 second to 50 seconds.

Figure 3 presents the results of the sensitivity experiments we conducted. In Figure 3, the x-axis shows the migration intervals in seconds we considered. For each application, the y-axis presents the normalized execution times for the considered migration intervals with respect to the execution times when migration is triggered at every 1-second.

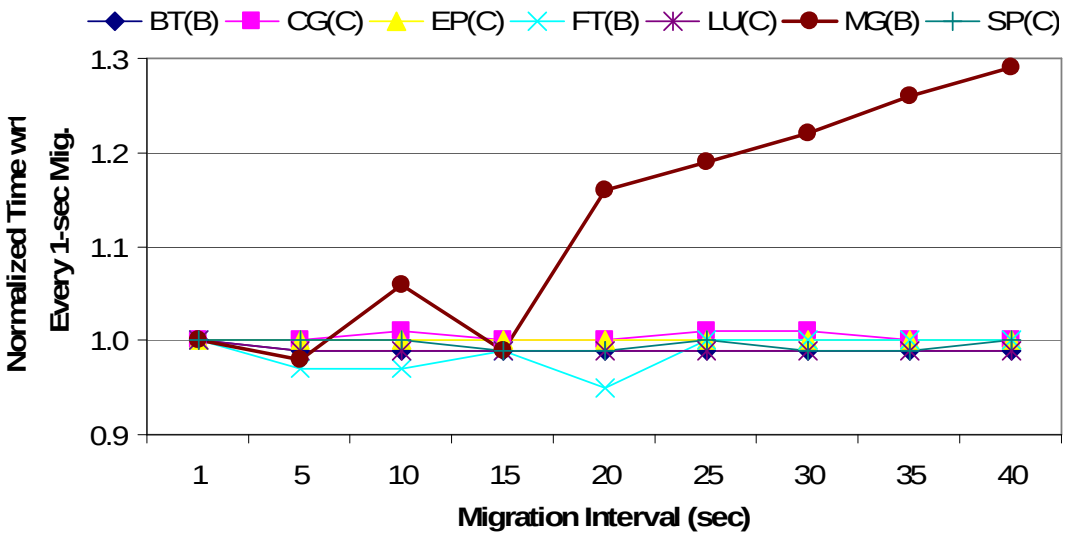


Figure 3: Sensitivity of the page migration scheme to the migration interval

Figure 3 shows that migration interval used does not have a major impact on the performance of the applications except MG. For MG, migration interval has a

significant impact due to the fact that MG is a short running program and when migration is triggered at a slower rate, MG does not benefit from page migrations. Thus, for our page migration experiments described in the remainder of this chapter, we chose to trigger page migration at every 5 seconds. We chose 5 seconds as the migration interval such that we would trigger enough number of migrations in MG to benefit from dynamic page migration but still keep a slower rate of migrations in the other applications for a lower migration overhead.

3.4.2. Reduction in Non-Local Memory Accesses

To quantify the benefits of our dynamic page migration approach, we counted the total number of non-local memory accesses for all applications with and without using dynamic page migration. We used the Sun Fire Link hardware monitors to measure the total number of non-local memory accesses in the applications.

Due to limitations in the number of accurate counters in the Sun Fire Link hardware monitor, we were not able to count the per board number of non-local memory accesses in an application during a single run. Instead, we ran each application once for each system board and counted the number of non-local memory accesses requested by the group of processors in that system board. We later calculated the total number of non-local memory accesses for an application as the sum of the non-local memory accesses for all system boards.

Table 4 presents the percentage reduction in the total number of non-local memory accesses when dynamic page migration is used compared to when memory pages are not migrated. In the second column, we give the total number of address transactions requested by each application during its execution. The third column

gives the percentage of non-local memory accesses without our page migration approach and the fourth column shows the percentage of non-local memory accesses when memory pages in the application are migrated using our dynamic page migration approach. The fifth column lists the percentage reduction in the total number of non-local memory accesses when dynamic page migration is used.

Table 4 shows that for all applications, our dynamic page migration approach was able to reduce the number of non-local memory accesses by 19.7-89.6% (The average reduction for applications is 58.3%).

Table 4 also shows that for MG, a significant number of non-local memory accesses were eliminated when memory pages were migrated. This is due to the fact that first-touch policy in the underlying operating system placed pages poorly in a single memory unit and our migration policy was able to migrate pages to several memory units according to their access pattern.

	# of Address Transactions (Millions)	Percentage of Non-local Accesses		% Reduction
		w/o Page Migration	Page Migration	
BT (B)	38,507	40.9	25.3	38.0
CG (C)	15,721	80.9	15.3	81.0
EP (C)	42	85.4	28.2	67.0
FT (B)	2,329	64.2	29.6	54.0
LU (C)	48,682	41.2	33.1	19.7
MG (B)	841	80.5	8.3	89.6
SP (C)	116,116	55.0	22.7	58.8

Table 4: Reduction in non-local memory accesses due to page migration

Unlike MG, for LU our dynamic page migration approach was not able to reduce the number of non-local memory accesses significantly. For LU, first-touch policy

placed memory pages better. Moreover, system boards uniformly access the majority of the memory pages that our dynamic approach was able to migrate. That is, while migrating those pages to a system board reduces the number of non-local memory accesses requested by the processors in that system board, the number of non-local memory accesses by the processors in all other system boards increases. Our dynamic page migration approach uses a simple decision mechanism that identifies the preferred location of a memory page as the system board that accesses it most. It does not take the access frequencies by other system boards into consideration. The access frequencies by other system boards may also be used to better decide whether a page should be migrated[71].

3.4.3. Impact of Page Migration on Cache Usage

The UltraSPARC III processors in the Sun Fire servers use physical addresses to index their external caches. Since page migration changes the physical addresses of the memory pages in an application, it is also necessary to ensure that our page migration approach does not have a significant impact on the cache usage of the applications. To quantify the cache usage of the applications, we counted the number of conflict and capacity misses (i.e. non-compulsory misses) during the execution of the applications with and without dynamic page migration. We counted the number of conflict and capacity misses in the applications using Sun Fire Link counters by measuring the number of write-back (WB) transactions requested. A WB transaction is requested when a dirty cache line is evicted from the external cache due to a capacity or conflict miss.

Table 5 presents the number of WB transactions with and without our page migration approach. Table 5 shows that our dynamic page migration approach does not significantly affect the number of conflict and capacity cache misses. It also shows that our dynamic page migration approach has a higher impact on EP compared to other applications. However, EP does not allocate a significant number of memory pages during its execution and thus the absolute number of cache misses is more than a factor of 20 lower than any other application we measured. Moreover, the total number of address transactions requested by EP is not significant due its effective use of local caches. The increase in cache misses in EP is mainly due to the invalidation of lines in caches caused by migration of memory pages.

	# of WB Transactions (Millions)		% Change
	w/o Page Migration	Page Migration	
BT (B)	14,948.8	14,900.1	-0.33
CG (C)	270.6	268.7	-0.67
EP (C)	12.3	12.6	2.38
FT (B)	855.0	851.8	-0.37
LU (C)	18,252.8	18,171.6	-0.44
MG (B)	217.4	218.0	0.28
SP (C)	39,223.3	39,139.9	-0.21

Table 5: Percent change in the number of write-back transactions

3.4.4. Execution Times

While reducing the number of non-local memory accesses in an application is important, what matters is the impact of this reduction on application's runtime. In this section, we look at the impact of our page migration approach on the execution

times of the applications. For each application, we conducted three different experiments and measured the total execution time for each experiment.

First, we ran each application using our dynamic page migration approach and measured the total execution time including overhead due to the creation of the helper threads and triggering memory page migrations. Even though the migration thread runs in parallel with other threads of the application, it suspends all application threads to trigger the actual page migrations and later resumes their executions. During the second set of experiments, we measured the original execution times of the applications with no intervention. Lastly, we conducted a third set of experiments to investigate the impact of binding application threads to fixed processors, and therefore the impact of dynamic page migration in isolation. During these experiments, we ran each application with page migration disabled but bound the threads to the processors in the system.

For each application and experiment, we repeated the experiment seven times and recorded the minimum of the execution times among all runs. We used the minimum execution time since we noticed higher variation in the original execution times for some applications. We suspect the higher variation in the original execution times of those applications is due to differences in the initial page placements and thread scheduling by the operating system.

Table 6 presents the execution times of the applications we analyzed. The second column lists the original execution times of the applications. In the third column, we present the execution times when the application threads are bound to the processors throughout the executions. The fourth column lists the execution times of the

applications when we migrate memory pages using our dynamic page migration approach. The fifth column presents the number of page migrations triggered. Lastly, the sixth column presents the overhead due to page migrations.

Table 6 shows that for all applications except LU and MG, when the application threads are bound to processors the applications run faster by 0.16-1.76% compared to their original executions. However, LU slows down by 0.6% where MG slows down by 2.2% when their threads are bound to the processors. Table 6 shows that binding application threads to the processors is almost always beneficial even though the performance gain is not significant.

	Execution Times (seconds)			# of Migrations	Overhead (seconds)
	Original	Bound Threads	Page Migration		
BT (B)	996	992	966	112,310	11.8
CG (C)	625	613	534	47,213	4.4
EP (C)	293	292	292	2,071	0.3
FT (B)	113	112	118	177,602	15.1
LU (C)	1981	1994	1978	132,696	13.1
MG (B)	31	32	26	49,884	2.7
SP (C)	3901	3854	3347	138,943	17.1

Table 6: Execution times, number of migrations and migration overhead

Table 6 also shows that the overhead due to page migration is mainly proportional to the number of page migrations requested and it ranges up to 12.8% compared to the original execution times of the applications. To guarantee that the migration thread touches the page next before all other threads, all other threads have to be suspended. If the operating system instead provided a system call that would allow applications to indicate the target locations of the memory pages, it would permit

migration of pages to their target locations during the next available opportunity, and thus reduce the page migration overhead.

Figure 4 presents the performance improvement when our page migration approach is used compared to both the original execution time and the execution time when the threads of the applications are bound to processors. Under the label of each application on the x-axis, Figure 4 also presents the migration overhead percentage with respect to the original execution time of the application. Figure 4 shows that our dynamic page migration approach was able to improve the execution performance of the applications except FT by up to 15.9% compared to their original executions. However, FT runs slower under our dynamic page migration approach.

Our dynamic page migration approach improved the performance of CG and SP by 14.5% and 14.2%, respectively, compared to their original execution times. CG and SP request many memory accesses and our dynamic page migration approach was able to eliminate many of the non-local memory accesses (see Table 4). In addition, dynamic page migration improved the execution performance of CG and SP by 12.8% and 13.2% respectively, compared to the executions where application threads are bound.

Like CG and SP, our dynamic page migration approach was also able to improve the performance of MG by 15.9% compared to its original execution time. Even though MG does not request many memory accesses, our page migration approach was still able to reduce the number of non-local memory accesses significantly (see Table 4). Compared to the execution of MG when its threads are bound to the

processors, dynamically migrating memory pages in MG improved the execution performance by 18.1%.

Figure 4 also shows that our dynamic page migration approach improved the execution performance of BT by 2.9% compared to its original execution and by 2.6% compared to execution where its threads are bound to processors. Figure 4 also shows that our page migration approach is not as effective for BT as for CG, MG, and SP, which is partially due to fact that the reduction in the number of non-local memory accesses in BT is not as high. Similarly, our page migration approach improved the performance of LU by 0.8%, which is also mainly due the small reduction in number of non-local memory accesses.

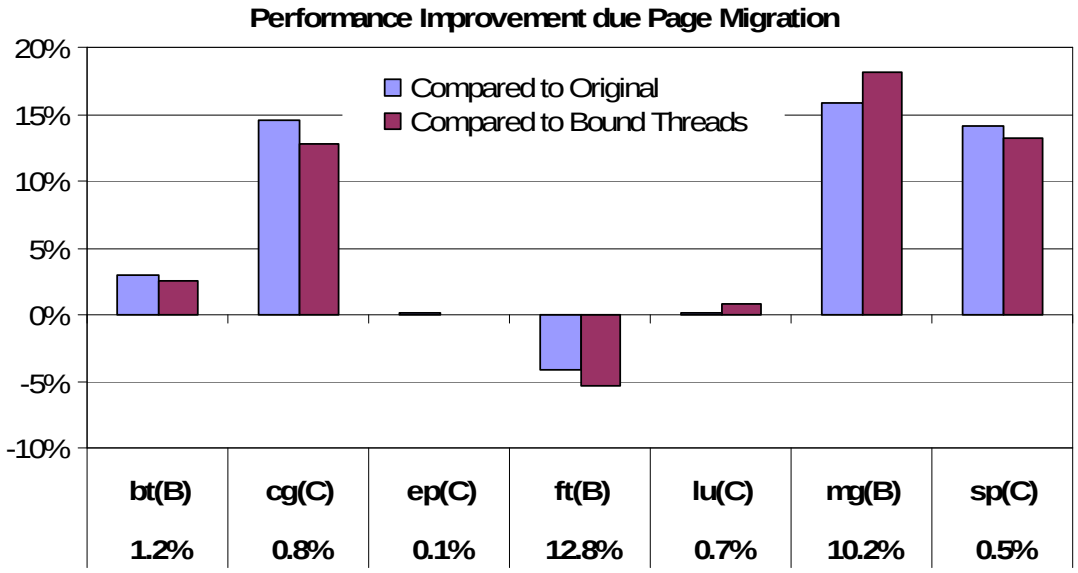


Figure 4: Performance gain for the applications under dynamic page migration

Figure 4 also shows that our dynamic page migration approach was not as effective in improving the execution performance of EP even though it reduced the number of non-local memory accesses by 67.0%. EP reuses data in the local caches of

the processors, and the majority of its memory accesses are requested at the beginning of its execution, before the memory pages are migrated.

Figure 4 shows that our dynamic page migration approach was not able to improve the execution performance of FT even though it reduced the number of non-local memory accesses in FT by 54.0%. Instead, our page migration approach slowed down the execution of FT by around 4.2% compared to its original execution. However, Figure 4 also shows that the slowdown for FT is mainly due to the overhead introduced by page migration, which is 12.8% of the original execution time for FT. That is, the reduction in the number of non-local memory accesses did not overcome the overhead introduced by migration of many pages that are initially placed poorly. Moreover, the page migration overhead for FT would be reduced significantly if the operating system did not require suspending application threads to trigger the actual migrations by touching pages and instead provided a mechanism to directly request migration.

3.5. Graphical User Interface

To visualize the page placement in the applications, we implemented a Graphical User Interface (GUI) that presents the locations of the virtual memory pages in terms of the memory units(boards) in the underlying CC-NUMA server. Our dynamic page migration GUI also presents additional information such as the number of page migrations triggered for each migration interval, the stack percentage bars indicating the percentages of pages migrated to each memory unit for the latest migration interval as well as since the application start.

Figure 5 shows the GUI snapshot for application MG when dynamic page migration is not used. The bottom window in Figure 5 visualizes the virtual address space of the application where each pixel (or a sequence of pixels when a portion of application's address space is displayed) represents a virtual page and the color of the pixel presents the memory unit the page is placed. The virtual page index increases from left to right and top to bottom, starting with the page index 0 at top left corner of the window. Note that in our GUI, there are 6 colors to represent the locations of the pages due to the fact the Sun Fire 6800 server we used have 6 memory units (boards).

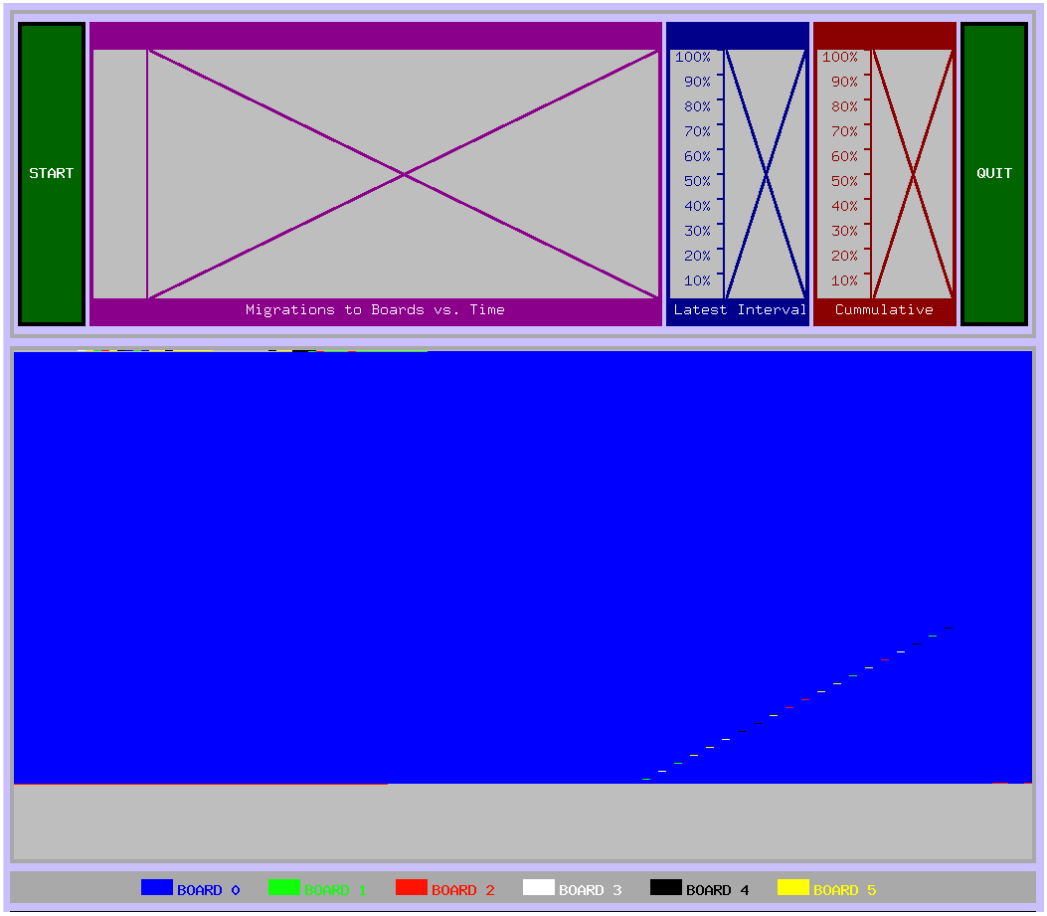


Figure 5: GUI snapshot for page placement in MG without page migration

Figure 5 shows that almost all of the memory pages in MG are placed in a single memory unit when MG is run without page migration. This is due to the fact that MG

starts with a single thread that initializes its data structures, hence first-touch placement in the underlying operating system places pages in the memory unit on the same board as the initialization thread runs.

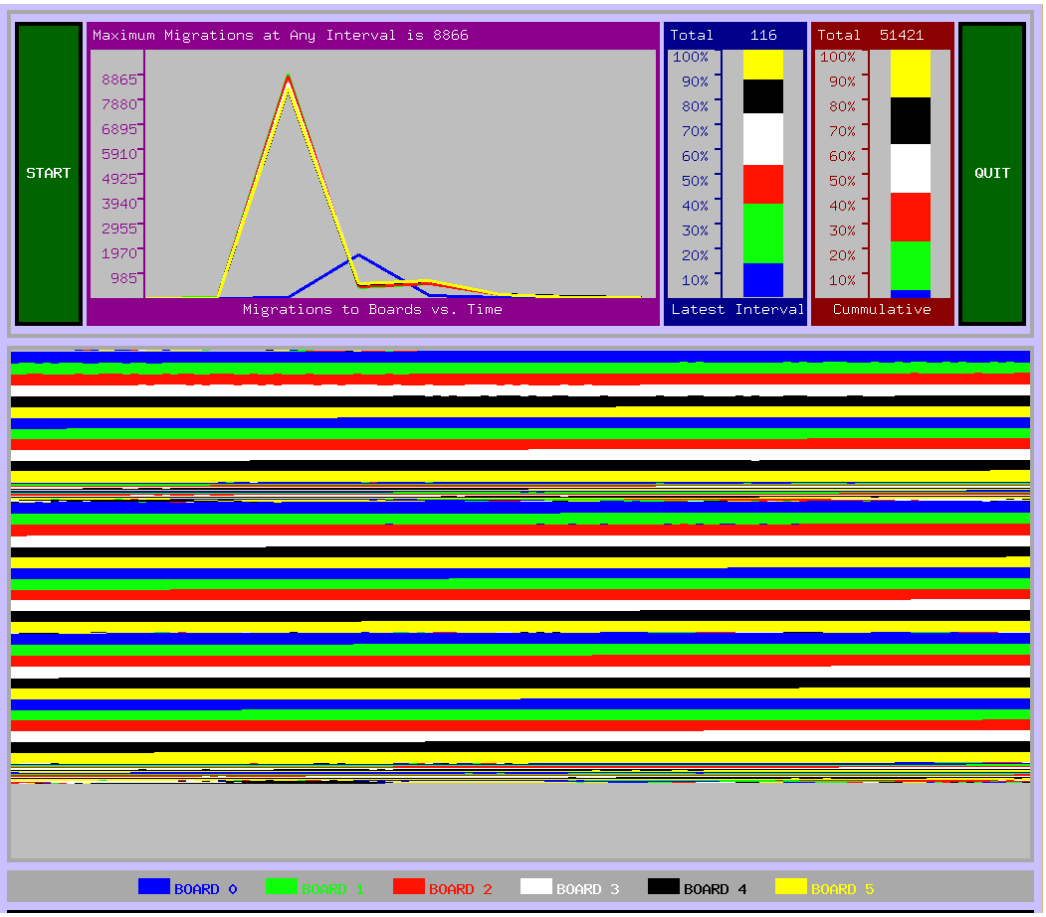


Figure 6: GUI snapshot for page placement in MG with page migration

Figure 6 shows the GUI snapshot for MG when run with dynamic page migration (every 5-second migration) after several migration iterations. Figure 6 shows that our dynamic page migration scheme was able to accurately migrate pages local to the processors accessing them most. Even though Figure 6 shows some imperfections in the placement of the pages due page migration, it clearly indicates the stride-access pattern in MG. We believe imperfections are caused by the fact that information on

some memory pages is not included in the profiles gathered from performance monitors due to use of interval sampling.

In addition to locations of the memory pages in an application, our dynamic page migration GUI also presents detailed information on page migrations triggered when run with dynamic page migration. The window at the top-left corner in Figure 6 displays the number of page migrations triggered to each memory unit for each migration interval. The top middle window displays the stack percentage bar that presents the percentages of migrations triggered to each memory unit for the latest migration interval in addition to the total number of pages migrated for the interval. Similarly, the top right window displays the stack percentage bar that presents the percentages of migrations triggered to each memory unit since the application start in addition to the total number of pages migrated.

3.6. Summary

In this chapter, we introduced an automatic profile-driven page migration scheme and investigated the impact of this page migration scheme on the memory performance of multithreaded programs. We used commercially available plug-in hardware monitors to profile the applications. We tested our dynamic page migration approach using the OpenMP C implementation of the NAS Parallel Benchmark suite.

Our dynamic page migration approach always reduced the total number of non-local memory accesses in the applications we analyzed compared to their original executions, by up to 90%. Our page migration approach was also able to improve the execution time of the applications up to 16% compared to their original executions.

We believe the effectiveness of our page migration approach also shows the advantage of putting the page migration policy at the user level while only relying on the operating system kernel to provide the actual migration mechanism.

We also believe that for page migration mechanism to be more beneficial, underlying operating system should provide means to trigger page migration without stopping the application. That is, if the user could simply request migration of a page and the underlying operating system could migrate the page during available idle cycles, most of the migration overhead would be hidden.

Chapter 4: Dedicated Monitors for Page Migration

In the user-level dynamic page migration scheme described in Chapter 3, we used the Sun Fire Link hardware monitors to sample access frequencies of memory pages by processors and consequently to identify preferred locations of memory pages at runtime. These monitors are centralized hardware and listen to the address bus of the underlying cc-NUMA system. They also provide a means to sample the address transactions from any device on the system interconnect. However, these types of hardware monitors are not always available for multiprocessor systems. Moreover, for non-bus based multiprocessors that do not use a common address and data bus, it is difficult to implement such centralized monitors to listen to the address transactions on the system interconnect.

Many processors have for some time included ways to count events such as cache misses, TLB misses, etc. Moreover, they provided ways to trigger an interrupt when a given number of events occur. More recently, processors such as Intel Itanium 2[37], provide the ability to capture the addresses involved in performance critical events including the address of an access that misses a cache or TLB in the memory subsystem. These monitors provide an opportunity to gather a sampled profile of page access behavior.

In this chapter, we investigate the use of several other potential sources of profiles gathered from hardware monitors in dynamic page migration and compare their effectiveness to using profiles from centralized hardware monitors that listen to the system interconnect. In particular, we investigate the effectiveness of using cache

miss profiles, TLB miss profiles from on-chip monitors, and the content of the processor TLBs.

We also introduce a simple hardware feature, called Address Translation Counters (ATC), which is specifically designed to gather profiles for dynamic page migration and compare its effectiveness with other sources of profiles. The ATC is a set of additional counters included in the TLBs of a processor and gathers accurate information on access frequencies to the memory pages by the processor.

To evaluate the effectiveness of each source of profiles in dynamic page migration, we conducted a simulation based study using a full system simulator. We present results of this study in terms of the number of page migrations triggered, reduction in the number of non-local memory accesses, and improvement in execution times of the applications. Similar to the page migration experiments described in Chapter 3, we present the results of our experiments for the applications in the OpenMP C implementation of NAS Parallel Benchmark suite.

4.1. Sources of Hardware Profiles for Dynamic Page Migration

In Chapter 3, we described how we used the centralized Sun Fire Link hardware monitors to identify the preferred locations of memory pages for dynamic page migration. In this section, we describe the other potential sources of profiles that can be used to generate page access frequencies for dynamic page migration.

4.1.1. Profiles Gathered from Distributed On-Chip CPU Monitors

Profiles of page access frequencies by processors in an application running on a cc-NUMA server can be gathered by using information about the cache or TLB misses

by each processor in the system. If the information about the number of cache or TLB misses on each page by a processor is known, the access frequency of the page by the processor can be approximated. However, for such information to be available, the addresses associated with the cache and TLB miss events are needed.

Many processors have for some time included hardware support to count interesting events for performance monitoring. Moreover, they have provided mechanisms to trigger an interrupt when a given number of events occur. More recently, an increasing number of processors provide the ability to capture the addresses involved in performance critical events. For example, the Itanium 2 processor provides a set of *event address registers* (EARs) that record the instruction and data addresses of data cache misses, the instruction and data addresses of data TLB misses, and the instruction addresses of instruction TLB and cache misses[37]. Thus, by distributed sampling of the addresses associated with the cache or TLB miss events, profiles of page access frequencies by processors can be generated. Moreover, since cache miss events are generally distributed throughout the execution and provide information on fine grain behavior, profiles of page access frequencies gathered from cache miss events may be more representative. Compared to cache misses, the number of TLB miss events is generally lower and these events may not correspond to the pages that are frequently accessed due to the fact that applications tend to keep frequently accessed pages in TLBs. In our research, we investigate the use of both cache miss and TLB miss information gathered from on-chip CPU hardware.

To access and control the hardware monitors, the publicly available instrumentation software such as *perfmon*[36] can be used. Perfmon is a standard kernel interface to access the hardware performance monitors available on all modern processors. In perfmon, the monitoring unit's state is encapsulated by a software abstraction called *context*, which is associated with a thread. For multi-threaded applications, it is necessary to create one context per thread and bind each application thread to a fixed processor. The perfmon interface also supports time based sampling and interval sampling where the sampling interval is expressed as a number of occurrences of an event. Our experiments in Chapter 3 showed that interval sampling produces more representative samples of all events in an application. Hence, we focus on interval sampling where samples are taken at fixed event boundaries.

4.1.2. Profiles Gathered from Valid Bit Information in TLB Entries

Hardware tries to keep virtual to physical page translation entries of the frequently accessed pages in the processor TLBs. That is, the content of the valid TLB entries in a processor potentially provides information on the pages that are mostly accessed by the processor. Thus, by sampling the content of the TLBs in a processor on a cc-NUMA server periodically, it is possible to approximate page access frequencies by the processor. Similarly, the information from each processor on a cc-NUMA server can be combined and page access frequencies by processors can be generated to guide page migrations in a dynamic page migration scheme.

To sample the content of valid TLB entries of a processor, the underlying operating system needs to provide a software sampling mechanism. In particular, the operating system needs to provide a means to query the list of valid entries and the

virtual addresses of the pages for each valid TLB entry. In our research, we assume the underlying operating system provides a system call that returns the list of virtual page addresses in the valid TLB entries for a given processor.

4.1.3. Address Translation Counters

To evaluate the effectiveness of sources of profiles in dynamic page migration, we designed a dedicated hardware monitor that gathers accurate page frequencies and compared the effectiveness of other sources of profiles with the dedicated monitors.

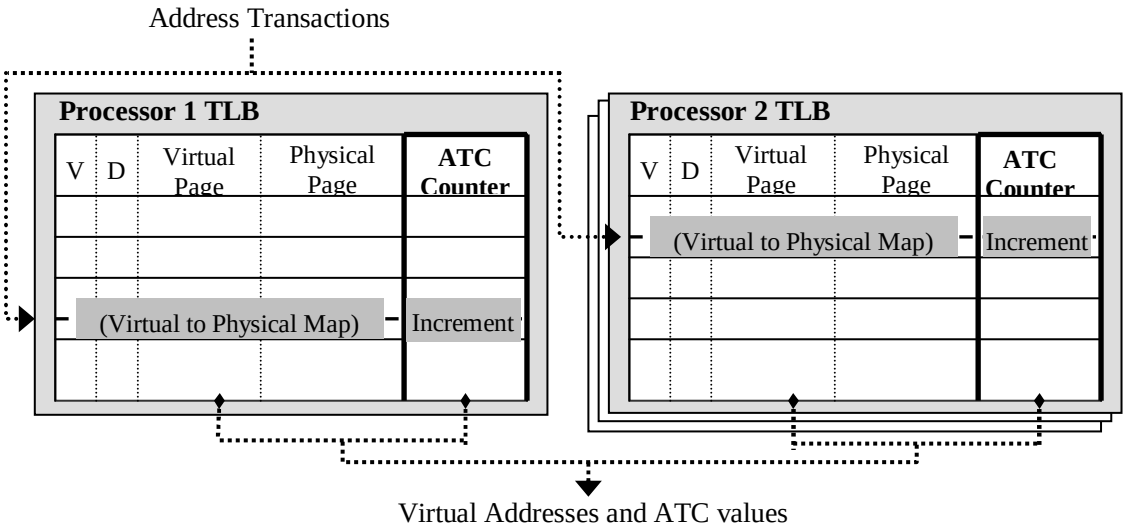


Figure 7: Information flow in the Address Translation Counters

The hypothetical hardware feature we use, Address Translation Counters (ATC), is a set of additional counters that is included in the TLBs of the processors. In ATC, a counter is included for each TLB entry in a processor (Shown in Figure 7) and incremented when a virtual to physical address translation is satisfied by the corresponding TLB entry. Moreover, when the content of a TLB entry is evicted due to a TLB miss or invalidated due to other reasons such as cache coherency operations, the counter associated with the TLB entry is cleared. The ATC is included in each

processor and will count the number of accesses to the memory pages by the processor using the virtual to physical address translations requested by the processor while the memory pages are actively accessed.

Information recorded by the ATC hardware can be gathered in several ways. One way is to sample the content of the counters regularly during execution along with the virtual page addresses associated with these counters. Another approach is that the operating system may provide low-overhead software traps such that when a TLB entry in invalidated due to a TLB miss or cache-coherency operation, the content of the corresponding ATC counter value and the associated virtual page address can be provided to the application (similar to the software TLB miss handler in MIPS processors[33]). Lastly, the underlying operating system could include an additional field for each page table entry where the TLB entry can be saved at context switches. Then, the TLB count information can be gathered via a system call by querying the page table content.

In our research, we assume the underlying operating system provides a system call to gather the information recorded by ATC in a given processor. To be used in dynamic page migration, the system call needs to return list of virtual addresses stored in each TLB entry and its corresponding ATC value. Moreover, for correct page access frequencies the counters need to be cleared to prevent repeated summation of counter values. Consequently, an application can periodically call the system call for each processor in a multiprocessor system and combine the information gathered from each processor to generate page access frequencies by each processor.

4.2. Simulation Framework

To evaluate the effectiveness of each source of profiles for page access frequencies in page migration, we conducted a simulation study using the full system simulator Simics [46], which is an efficient system level instruction set simulator. For our research, we chose to simulate a Sun Fire 6800 as the target cc-NUMA system. Despite its small ratio of local to remote memory latency, it allows us to compare our simulation study to the actual page migration scheme in Chapter 3.

To simulate a target machine on Simics, it is also necessary to install the corresponding operating system to run on the simulated machine. We installed the Solaris 9 (Generic_117171-07) binaries on the simulated machine.

Sun Fire systems are built from UltraSparc III processors[14,68]. The memory subsystem in UltraSparc III processors includes five caches, four on-chip and one external. These caches include an L1 data cache, an L1 instruction cache, a prefetch cache, a write cache and an L2 external cache. In addition, the memory management unit in UltraSparc III processors includes two data and two instruction TLBs that are accessed parallel. In each pair of TLBs, one TLB is smaller and is used to support larger page sizes (64K-4M) efficiently. When 8K pages are used, the smaller TLB is included as part of the larger TLB.

By default, Simics does not model any cache system or memory subsystem. It uses its own internal memory representation where the memory is always up to date with the latest CPU and device transactions[73]. However, the functionality of Simics can be extended by user-written modules[72]. Simics provides an interface for users to provide modules to observe and modify the behavior of the transactions that go

through the memory system. This interface is composed of two different interfaces acting at different phases of a memory transaction execution. In particular, the timing-model interface provides access to a transaction before it is executed (i.e., it has just arrived at the memory-space). The timing model interface can also be used to change the timing and the execution of a memory transaction, as well as to modify the value of a store going to the memory. The snoop-memory interface provides access to a transaction after it has been executed.

To simulate the memory subsystem of UltraSparc III processors, we both modified the already available Simics modules and implemented a new timing module. We also implemented a separate module to simulate the on-chip TLBs.

In addition to memory subsystem, we also implemented a monitoring module for the data collection methods we want to evaluate. These include on-chip hardware performance monitors to gather cache miss and TLB miss information, the centralized Sun Fire Link monitors to gather interconnect transactions, and our hypothetical hardware feature ATC to gather page access frequencies. Figure 8 shows the overall framework of the memory subsystem model we used in our simulation study.

The *transaction filter* in Figure 8 filters the transactions according to whether they are memory or device transactions as well as their cacheability status. For our simulation, we only focus on cacheable memory transactions. These transactions are passed to the *type filter*, which separates instruction and data accesses and sends them to separate L1 caches. Since accesses can cross a cache-line boundary, the *line splitters* are connected before L1 caches to let correctly aligned accesses go through untouched whereas unaligned accesses are split in two accesses.

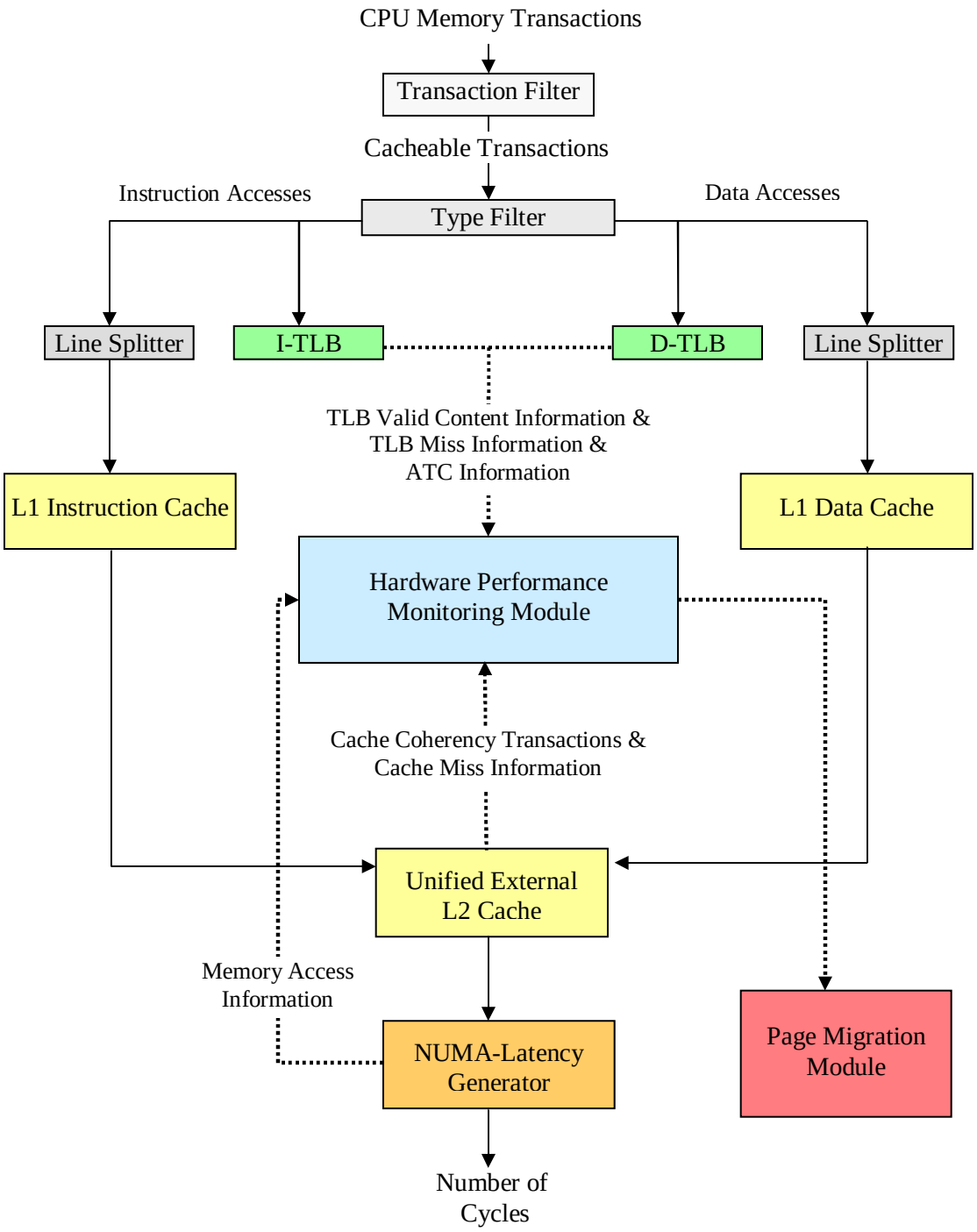


Figure 8: Simulation framework for the memory subsystem in each processor

Simics comes with two different cache models. The *g-cache* module is the standard cache model. It handles one transaction at a time and cache operations are performed in order. The cache returns the sum of the stall times reported for each

operation. The *g-cache-ooo* module is a more complex model adapted to Simics with micro architectural interface extensions. It handles multiple outstanding transactions and keeps track of their current status. g-cache-ooo is the standard cache model for Simics out-of-order execution processors[72].

In this research, we only simulate L1 and L2 caches from the memory subsystem due to the difficulty of simulating other caches and their interactions accurately. To simulate the L1 caches and unified L2 cache, we modified the g-cache module available in Simics distribution. Even though UltraSparc processors are out-of-order processors, we used the g-cache module rather than g-cache-ooo for our simulation due to fact that cache coherency protocols are only implemented in the g-cache module. Moreover, since Sun Fire servers use a modified version of the MESI protocol for cache coherency, called MOESI, and the g-cache module implements the MESI coherency protocol, we modified the g-cache module to add necessary modifications to handle additional states and state transitions in the MOESI protocol.

To simulate the on-chip TLBs, we implemented a separate Simics module. This module is configurable where the number of lines, associativity, page size and replacement policy are given via configuration scripts. We also implemented the ATC hardware in this module by including additional hardware counters in the TLB entries for each locality group in the simulated machine.

The *Page Migration* module in Figure 8 implements the dynamic page migration algorithm described in Section 3.2. That is, it gathers profiles to guide the page migrations via sampling the information from hardware monitors and generates page access frequencies. It also triggers page migrations at fixed time intervals. This

module is also configurable in the sampling interval for hardware monitors, simulated seconds between migrations and the type of hardware monitors and source of profiles that will be used for page migration.

The *NUMA-Latency Generator* module generates stall times for local or non-local access latency for each access request that misses the caches in the memory subsystem. This module simulates the actual memory accesses requested during application execution including the accesses due cache misses, write-backs due to eviction of dirty cache lines, or writes on write-through caches. This module also provides information on the locality of each access to the hardware performance monitoring unit to measure the percentage of memory access locality during application execution.

4.3. Simulation Experiments

To investigate the effectiveness of each source of profiles in dynamic page migration, we conducted simulation experiments using the OpenMP C implementation of the NAS Parallel Benchmark suite. We chose applications with different sizes from A to B (moderate to large data set sizes) such that each application would have a similar memory footprint. Moreover, we modified the number of iterations in each application to keep the simulation time manageable. We compiled the applications using Sun's native compiler, Sun C 5.5 EA2, with optimizations (-O3 -xopenmp=parallel) on an actual Sun Fire 6800 server and copied them to the simulated Sun Fire machine. For all experiments, as the target machine, we booted a 24-processor Sun Fire 6800 with 12GB of main memory where each locality group contains 2GB main memory. The default processors in the simulated machine are

75MHz UltraSPARC III. Even though the clock frequency of a simulated processor can be set arbitrarily in Simics without any affect on the actual speed of simulation, the default frequency is set to a lower value to emulate an interactive system such that keyboard and mouse input events in the simulated machine are handled with short delays. Since we used interactive mode to set up the simulated machine including installing the operating system, copying the compiled executables to the simulated disk, we used the default processor settings.

To quantify the benefits of using each source of profiles on dynamic page migration, we ran a series of experiments with and without page migration. For each application, we ran the application with dynamic page migration several times varying the source of profiles. Additionally, to investigate the impact of accurate page access frequencies on the effectiveness of dynamic page migration, we also ran each application and migrated pages based on perfect profiles. Perfect profiles are gathered by having the simulator use full memory access history for all page references to allow us to quantify the cost of the less than perfect profiles produced by our sampling techniques.

For the experiments with page migration, we triggered page migration at every 50 simulated seconds. Even though we chose migration interval as 5 seconds in our dynamic page migration studies on the actual machine, we chose to trigger page migrations at every 50 seconds for our simulation study due to the fact that the simulated clock cycle is 10 times slower compared to the actual machine.

For all experiments, we used the same simulation parameters for the simulated memory subsystem except we varied the sampling method used to gather profiles

from hardware monitors. Table 7 summarizes the parameters and their values we used in our experiments for each source of profiles to generate page access frequencies.

	Interconnect Transactions	Cache Misses	TLB Misses	TLB Content	ATC Content
Sampling Method	Centralized	Distributed			
Sampling Interval	Every 512 transactions	Every 512 miss events		Every 16K translation events	
Local Latency	225ns				
Non-Local Latency	300ns				
I-TLB	128-entry, 2-way associative, 8K pages				
D-TLB	512-entry, 2-way associative, 8K pages				
L1 D- Cache	64 KB, 4-way associative, 32-byte lines, 2ns hit time				
L1 I-Cache	32KB, 4-way associative, 32-byte lines, 2ns hit time				
L2 Cache	8MB, 2-way associative, 512-byte lines, 16ns hit time				

Table 7: System parameters and their values used in simulation experiments

4.3.1. Memory Access Locality Experiments with Page Migration

For each simulation experiment, we measured the percentage reduction in the number of non-local memory accesses in the application when memory pages are dynamically migrated compared to its original execution. We also measured the total number of pages migrated throughout the execution of the application.

Table 8(a) presents the percentage of non-local memory accesses for the applications we tested with and without page migration. The second column presents the percentage of non-local memory accesses in the original executions. The next five columns present the percentages of non-local memory accesses when applications are run with dynamic page migration using different sources of profiles to generate page access frequencies. The last column presents the percentage of non-local memory accesses using accurate page access frequencies gathered from all actual memory

accesses. For each application and source of profiles, Table 8(a) also gives the percentage reduction in the number of non-local memory accesses with respect to the original execution of the application.

Like Table 8(a), Table 8(b) presents the number of page migrations triggered when applications are run with dynamic page migration using different sources of profiles to generate page access frequencies. However, for each application, the number in parenthesis in each cell in Table 8(b) gives the ratio of the number of page migrations triggered with respect to the number of page migrations triggered using perfect profiles. We present these ratios for a better comparison of the number of page migrations triggered for different source of profiles.

At first glance, it looks like dynamic page migration is effective in reducing the number of non-local memory accesses independent of the source of profiles used to gather page access frequencies. The one exception is that the number of non-local memory accesses is increased for LU only when TLB miss information is used. Overall, the reduction in the number of non-local accesses ranged from -9.6% to 87.3%. Moreover, it appears that the behavior of the different data collection techniques can broadly be grouped in to three different groups based on the number of non-local memory accesses and the number of page migrations triggered. In particular, the results show that using interconnect transactions performs similar to using cache miss information, and using ATC content performs similar to using TLB content, and using TLB miss information performs poorly compared to other sources of profiles.

	Orig. % Non-Local Accesses	% of Non-Local Accesses (% Reduction compared to the Original Execution)					
		Intercon. Trans.	Cache Misses	TLB Misses	TLB Content	ATC Content	Perfect Profiles
BT-A	42.0	29.4 **(30.1)**	29.1 **(30.9)**	36.9 **(12.1)**	30.7 **(26.9)**	28.2 **(33.0)**	28.1 **(33.3)**
CG-B	79.5	13.7 **(82.7)**	11.9 **(85.0)**	26.6 **(66.5)**	11.8 **(85.2)**	11.3 **(85.8)**	11.7 **(85.3)**
FT-B	77.0	66.5 **(13.6)**	66.0 **(14.3)**	71.9 **(6.6)**	64.2 **(16.7)**	64.0 **(16.9)**	63.8 **(17.1)**
LU-B	42.5	35.5 **(16.5)**	34.3 **(19.3)**	46.6 **(-9.6)**	42.2 **(0.8)**	41.1 **(3.3)**	33.5 **(21.3)**
MG-B	80.6	14.8 **(81.7)**	12.9 **(84.0)**	46.3 **(42.6)**	10.3 **(87.2)**	10.2 **(87.3)**	10.0 **(87.6)**
SP-B	69.0	54.0 **(21.7)**	53.8 **(22.0)**	62.6 **(9.3)**	55.8 **(19.1)**	54.5 **(21.0)**	52.9 **(23.4)**

(a) Percentage of non-local accesses for different sources of profiles

	Number of Page Migrations Triggered (Ratio wrt. Perfect Profiles)					
	Intercon. Trans.	Cache Misses	TLB Misses	TLB Content	ATC Content	Perfect Profiles
BT-A	34,529 **(2.20)**	31,422 **(2.00)**	36,472 **(2.32)**	17,298 **(1.10)**	14,122 **(0.90)**	15,730
CG-B	18,828 **(0.97)**	18,920 **(0.98)**	18,524 **(0.96)**	19,823 **(1.02)**	19,308 **(1.00)**	19,344
FT-B	190,313 **(1.05)**	214,605 **(1.18)**	98,320 **(0.54)**	156,180 **(0.86)**	155,578 **(0.86)**	181,632
LU-B	22,881 **(2.23)**	21,177 **(2.07)**	19,492 **(1.90)**	8,589 **(0.84)**	4,897 **(0.48)**	10,241
MG-B	51,361 **(1.06)**	52,435 **(1.08)**	34,009 **(0.70)**	49,102 **(1.01)**	48,552 **(1.00)**	48,397
SP-B	35,420 **(1.43)**	34,453 **(1.39)**	40,571 **(1.64)**	25,035 **(1.01)**	25,233 **(1.02)**	24,814

(b) Number of page migrations triggered for different sources of profiles

Table 8: Results of memory locality experiments for different sources of profiles

Table 8(a) shows that when perfect profiles are used, dynamic page migration reduces the number of non-local memory accesses in the applications by 17.1-87.6%. Table 8(a) also shows that dynamic page migration using interconnect transactions reduces the number of non-local memory accesses by 13.6% to 82.7%. For some applications the reduction in the number of non-local memory accesses is slightly lower compared to the reduction percentages presented in Chapter 3. This is mainly due the fact that we modified the number of iterations in the applications to obtain manageable simulation times. Most of the page migrations are triggered early in the execution of these applications and during the rest of the execution they benefit from these page migrations. Thus, by reducing the number of iterations in an application, the application does not fully benefit from dynamic page migrations. However, Table 8(a) also shows that the reduction using interconnect transactions are comparable to using perfect profiles, which indicates that using interconnect transactions in dynamic page migration is effective in approximating the actual page access frequencies by processors.

Table 8 (a) and (b) show that using cache miss information in dynamic page migration performs slightly better compared to using interconnect transactions in terms of the reduction in the number of non-local memory accesses and the number of page migrations triggered. For the majority of applications, using cache miss information reduces the number of non-local memory accesses slightly more and triggers slightly fewer page migrations compared to using interconnect transactions. Moreover, the results show that using cache miss information performs closer to using perfect profiles compared to using interconnect transaction in terms of the

reduction in the number of non-local memory accesses. Thus, by distributed sampling of cache miss information from on-chip CPU hardware monitors in a multiprocessor server, dynamic page migration can accurately generate page access frequencies in the applications and can be as effective as sampling interconnect transactions via centralized hardware counters.

Table 8 (a) and (b) also show that using TLB and ATC content in dynamic page migration perform similar and they are comparable in terms of the reduction in the number of non-local memory accesses to using cache miss and interconnect transaction information for all applications except LU. In LU, they are not as effective in reducing the number of non-local memory accesses even though they trigger significantly fewer page migrations. In terms of the number of page migrations triggered, using TLB and ATC content tend to trigger fewer page migrations compared to using cache miss information and interconnect transactions. However, for CG and MG where dynamic page migration is highly effective, they trigger comparable number of page migrations.

Table 8 (a) and (b) also show dynamic page migration using TLB miss information is not as effective as other sources of profiles. Even though using TLB miss information triggers fewer migrations, it is not as effective in reducing the number of non-local memory accesses because the page access frequencies gathered from TLB miss information is not representative of page access frequencies in the applications. Moreover, dynamic page migration using TLB miss information increases the number of non-local memory accesses for LU by around 10%.

Overall, Table 8 (a) and (b) show that the sources of profiles other than using TLB miss information perform similar in terms of the reduction in the number of non-local memory accesses. More importantly, they show that cache miss profiles gathered from on-chip hardware monitors, which are typically available in current micro-processors, can be effectively used to guide dynamic page migration in an application. This is particularly encouraging since such on-chip counters are included in many recent processors, and instrumentation software to access these counters are publicly available. Thus using cache miss information via distributed sampling in dynamic page migration is an easy and effective approach. Even though using TLB and ATC content performs slightly better for some applications, their use requires new hardware and new system calls in the operating.

4.3.2. Case Study: Memory Access Locality in MG

To further investigate how dynamic page migration works using different sources of profiles to generate page access frequencies, we present the change in memory access locality versus time during execution of MG (size B). We also present the number of page migrations triggered versus time for MG. We chose to present the results for MG due to the fact that both our actual dynamic page migration approach in Chapter 3 and our simulation study have shown migration to be most effective for MG.

Figure 9 presents the number of page migrations triggered versus time in MG for experiments with dynamic page migration using different sources of profiles. Similarly, Figure 10 presents the percentages of non-local memory accesses versus time in MG with dynamic page migration using different sources of profiles as well as without migration. We measured the percentage of non-local memory access and

the number of page migrations after each page migration interval, thus the x-axis in both figures is labeled with increasing order of migration interval.

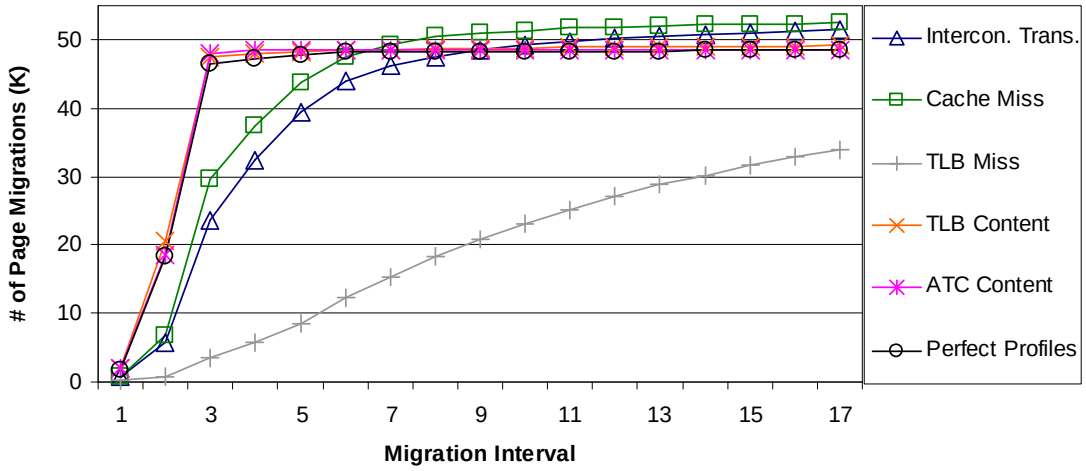


Figure 9: Number of migrations triggered by time in MG

Figure 9 shows that when MG is run with dynamic page migration using sources of profiles other than TLB miss information, the majority of the page migrations are triggered early in the execution. Using TLB and ATC information triggers more migrations during the first seven migration intervals compared to using interconnect transactions and cache miss information but the latter sources of profiles trigger slightly more migrations in total. Overall, the number of migrations triggered in MG is comparable for all sources of profiles other than using the TLB misses. Dynamic page migration using TLB miss information triggers page migrations throughout the execution and triggers significantly fewer overall page migrations.

More importantly, Figure 9 shows that using profiles other than TLB miss information triggers a similar number of page migrations compared to using perfect profiles. However, using TLB and ATC content matches the behavior of using perfect profiles slightly better compared to using other sources of profiles.

Figure 10 shows that the percentage of non-local memory accesses in the original execution of MG increases steeply during the first third of the execution and increases slowly during the rest of execution. Figure 10 also shows that when run with dynamic page migration, the percentage of non-local memory accesses starts to decrease after the third migration interval for all sources of profiles except using TLB miss information. Using TLB and ATC content reduced the number of non-local accesses more during first couple of migration intervals compared to using cache miss and interconnect transactions. However, towards the end of the execution the difference in percentage of non-local memory accesses is reduced. Overall, using TLB and ATC content matches to the behavior of using perfect profiles slightly better in MG. Unlike other sources of profiles, Figure 10 shows that using TLB information is not as effective for MG and the percentage of non-local memory accesses starts to decrease later during execution compared to other sources of profiles, which results in significantly smaller reduction in the number of non-local memory accesses.

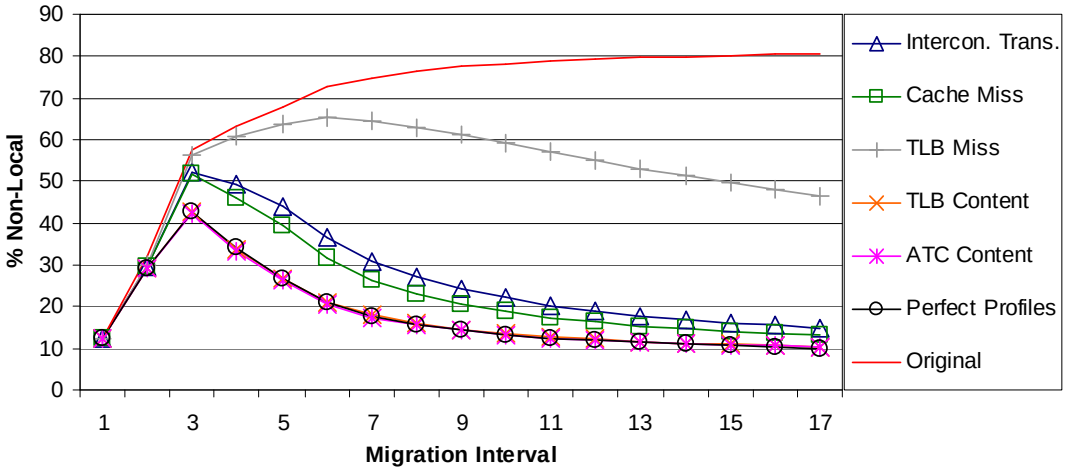


Figure 10: Percentage of non-local memory accesses by time in MG

Overall, the figures show that using TLB content and ATC content is initially more effective in reducing the number of non-local memory accesses in MG compared to using cache miss information and interconnect transactions. However, by the end of execution, all sources of profiles except TLB miss information provide comparable information.

4.3.3. Execution Times

To investigate the impact of the reduction in the number of non-local memory accesses in the execution times of the applications, we also measured the total number cycles spent to execute each application using different sources of profiles in the simulator. For each simulation experiment, we measured the total number of cycles spent to satisfy memory accesses.

Our simulation experiments showed that even though the total number of cycles to satisfy memory accesses is reduced by up to 16% for the applications, the impact of this reduction on the total number of cycles to execute the applications was not significant (typically about a 0.5% improvement). This is due to the fact that even though our simulator can simulate instructions executed by an application accurately, it lacks the ability to properly simulate the contention for the memory units. Moreover, in such a simulation environment it is also difficult to accurately simulate simultaneous out-of-order issue of multiple instructions by multiple processors. Thus, since our workload exhibits very low cache miss behavior, the performance improvement in actual memory accesses did not have a significant impact on the overall performance of the applications when the simulator executes one instruction at a time. We believe this simulation limitation contributes to conclusions reached by

previous researches that indicated limited benefit to page migration in cc-NUMA systems[11,50,71].

To verify this claim, we conducted experiments where we ran MG (size B) under different page placement scenarios on an actual machine to isolate the impact of memory contention and the impact of the reduction in the number of non-local memory accesses on its execution performance.

Originally, almost all pages in MG are placed in to a single memory unit which results in 80.5% of non-local memory accesses in its execution. Since the majority of the pages are placed on a single memory unit, the contention to this memory unit during execution is very high. To investigate the impact of memory contention on the execution performance of MG, we ran MG under two different page placement settings on an actual Sun Fire 6800 server. In one setting, its memory pages are placed uniformly on all memory boards and in the other setting, all of its pages are placed in a single memory unit. We measured the percentage of non-local memory accesses in MG for each run using the Sun Fire Link counters and the fraction of non-local memory accesses remained around 80%. Our experiments show that by placing the pages in MG uniformly to all memory boards, the execution performance of MG improved by 10.2% compared to its execution where all pages are placed in a single memory unit. This indicates that simply reducing the memory contention to a single memory unit improved execution performance of MG improved by 10.2%. Thus, the lack of ability to accurately capture such memory contention in the simulated machine partially explains why our simulation experiments did not yield improvement in the total number of cycles executed by the applications.

To investigate the impact of the reduction in the number of non-local memory accesses on the execution performance, we also ran MG on the actual Sun Fire server and placed all pages at their preferred locations at the beginning of the execution. We did this placement based on the data gathered using our dynamic page migration scheme described in Chapter 3. Our experiments showed that the execution performance of MG improved by 10.3% when the pages are placed in their preferred locations compared to when all pages are placed uniformly over all memory units. This improved placement resulted in an 81% reduction in the number of non-local memory accesses. The difficulty to accurately simulate the actual memory subsystem and latency hiding in the instruction executions using in-order cache modules in the simulator also partially explains why our simulation experiments did not yield an improvement in the total number of cycles executed in the applications despite the reduction in the number of non-local memory accesses in the applications.

To better evaluate the impact of the reduction in the number of non-local memory accesses on the execution times of the application, we adopted a different approach. In this approach, during simulation experiments, we recorded the actual page migrations triggered in each simulation experiment in to a log file and used the log file to trigger page migrations on an actual machine using our dynamic page migration scheme described in Chapter 3. To do this, we modified our page migration system not to gather profiles from hardware monitors but instead use log files generated during simulation to guide page migrations. For each application and source of profiles, we recoded the page migrations and ran the application on an

actual Sun Fire 6800 server where page migrations are triggered at fixed time intervals using the recoded migration entries.

Table 9 presents the percentage improvement in the execution times of the applications we tested with and without page migration on an actual Sun Fire 6800 server compared to their original execution times. Starting with the second column, the next five columns present the percentage improvement in the applications when they are run with dynamic page migration using different sources of profiles to generate page access frequencies. The last column presents the percentage improvement using accurate page access frequencies gathered from all actual memory accesses. For each application and source of profiles in Table 9, the positive improvements in the execution performance of applications are shown in bold.

	% Improvement in Execution Times compared to Original					
	Intercon. Trans.	**Cache Misses**	**TLB Misses**	**TLB Content**	**ATC Content**	**Perfect Profiles**
BT-A	-1.04	-0.76	-1.70	**0.22**	**0.83**	**0.89**
CG-B	**8.35**	**8.38**	**6.51**	**8.53**	**8.63**	**8.48**
FT-B	-0.22	-1.82	-0.08	-2.35	-1.25	-1.46
LU-B	-0.79	-1.01	-0.85	-0.08	**0.36**	**0.26**
MG-B	**16.47**	**15.84**	**13.06**	**18.09**	**18.28**	**17.97**
SP-B	**5.47**	**5.77**	**2.77**	**7.11**	**7.55**	**7.15**

Table 9: Improvement in execution performance for different sources of profiles

At first glance, Table 9 shows that dynamic page migration is effective at improving the execution performance of applications CG, MG and SP independent of the source of profiles used to gather page access. Dynamic page migration slightly slowed down the execution of FT for all sources of profiles and slowed down the execution of BT and LU for profiles other than TLB and ATC information. The improvement in runtime for all applications ranged from -1.82% to 18.28%.

Moreover, similar to Table 8 (a) and (b), Table 9 shows that the behavior of dynamic page migration can broadly be grouped in to three different groups in terms of the improvement in the execution times of the applications. That is, Table 9 shows that using interconnect transactions performs similar to using cache miss information, and using ATC content performs similar to using TLB content, and using TLB miss information performs differently compared to other sources of profiles. Also with our ATC hardware, we were able to improve the execution times of 5 of 6 applications versus 3 of 6 for our original centralized monitors.

Table 9 shows that when perfect profiles are used, dynamic page migration improves the execution performance of the applications by -1.46-17.97%. Table 9 shows that for applications CG, MG and SP, dynamic page migration improves their execution performance independent from the source of profiles used compared to their original execution. The improvement is up to 18.28% for MG. Moreover, for these applications, using TLB content and ATC information performs slightly better compared to using interconnect transactions and cache miss information. This is due to the fact that using TLB content and ATC information tend to trigger fewer page migrations, which results in less migration overhead. More importantly, for these applications using cache miss information performs comparable if not better than using interconnect transactions. The minor differences for these applications when interconnect transactions and cache misses are used are due the differences in the reduction of the number of non-local memory accesses and the number of page migrations triggered during the execution of these applications.

Table 9 shows that using TLB content and ATC information performs slightly better for the applications BT and LU compared to using other profiles. This is due to the fact the for these applications dynamic page migration triggers fewer page migrations compared to other profiles while producing a similar reduction in the number of non-local memory accesses. This is due the fact that the overhead due migration of pages in not as much when TLB content and ATC information is used for these applications.

Table 9 shows that for FT none of the profiles was effective in improving the execution performance. This due to the fact that the reduction in the number of non-local memory accesses for FT is only around 15% and the overhead introduced by migrations of any pages did not overcome the benefits due to improvement in memory access locality of FT.

Table 9 also shows that even though using TLB miss information triggers fewer migrations, it is not as effective in reducing the number of non-local memory accesses indicating that the page access frequencies gathered from TLB miss information is not representative of page access frequencies in the applications.

4.4. Summary

In this chapter, we evaluated the effectiveness of using of several potential sources of hardware profiles in dynamic page migration and compared their effectiveness to using profiles from centralized hardware monitors. In particular, we investigated the effectiveness of using profiles gathered from on-hip CPU monitors, the content of the processor TLBs and a hypothetical hardware feature designed specifically for dynamic page migration.

Our experiments showed that the reduction in the number of non-local memory accesses in the applications ranges up to 87.3% compared to not using page migration, which resulted in up to an 18.3% improvement in execution time. Moreover, our experiments showed that using interconnect transactions performs similar to using cache miss information, and using ATC content performs slightly better still. However, using TLB miss information performs poorly compared to the other sources of profiles.

More importantly, our experiments showed that using cache miss information performs comparable to using profiles gathered from hardware monitors specifically designed for page migration as well as perfect profiles constructed from all actual memory accesses. That is, our experiments demonstrated that cache miss profiles gathered from distributed on-chip hardware monitors, which are typically available in current micro-processors, can be effectively used to guide dynamic page migrations in an application.

Chapter 5: Inadequacy of Page Level Optimization

The Java Programming Language is gaining popularity on multiprocessor servers[58]. Moreover, in recent years, increasing interest is being shown in the use of Java Programming Language for client-server computing. Java server applications often need to perform several tasks at the same time to satisfy the requests of several client applications. To facilitate parallelism, server applications are usually multithreaded. Due to the large memory requirements and multi-threaded nature of the Java server applications, they are gaining importance as a workload for commercial shared-memory multiprocessor servers.

Prior research[43,50,71,75] has shown that dynamic page placement techniques on cc-NUMA systems are most effective for applications with regular memory access patterns, such as scientific applications. In these applications, large static data arrays that span many memory pages are divided into segments and distributed to multiple computation nodes where only one or a few computation nodes accessing each data segment most.

However, unlike scientific applications, Java programs tend to make extensive use of heap-allocated memory and typically have significant pointer chasing [58]. Thus, unlike scientific applications, dynamic page placement techniques may not be as beneficial for Java applications since they allocate many objects, with different access patterns, on the same memory page. Since the page placement mechanism used in the operating system is transparent to the standard allocation routines, the same memory page can be used to allocate many objects that are accessed by different processors.

Due to Translation Lookaside Buffer size issues, cc-NUMA servers tend to use super pages of several megabytes, which further increases the likelihood of allocating objects that have different access patterns on the same memory page. As a result, to better optimize memory access locality in Java applications running on cc-NUMA servers, heap objects should be allocated or moved so that objects that are mostly accessed by a processor whose memory is local to that processor.

In this chapter, we first evaluate the potential of existing well-known locality optimization techniques and present the results of a set of experiments where we applied dynamic page migration to a Java server application. In our experiments, we used the dynamic page migration scheme described in Chapter 3.

We next introduce an approach to measure the memory behavior of Java server applications at the object level. Our approach is based on source code instrumentation of the underlying virtual machine to gather information about the heap allocations and sampling the address transactions during the execution of the application via hardware performance monitors.

Lastly, we present the results of a simple optimization technique that tries to reduce the number of non-local memory accesses in the young generation using the *madvise* system call in Solaris 9. We used the move-on-next-touch feature of the madvise system call on the pages in the young generation to move the pages local to the processor that touches them next.

5.1. Software Components

In this section, we describe the software components used in our research. We first briefly describe the original memory management used in the Hot Spot Server

VM[67]. We next describe the SPECjbb2000 benchmark[64] we used as a typical server application to evaluate our techniques described in this chapter.

5.1.1. Java HotSpot Server VM (version 1.4.2)

It is commonly accepted that the majority of the objects in Java applications die young[40]. That is, most of the objects can be reclaimed shortly after being allocated. For instance, *Iterator* objects are often alive for the duration of a single loop. Some objects however do live longer. For instance, there are typically objects allocated at program initialization and they live until the program terminates. Between these two extremes, there are objects that live for the duration of some intermediate computation. Even though distribution of objects with different lifetimes vary from one application to another, most applications contain many short-lived objects[67].

For efficient garbage collection, the Java HotSpot VM exploits the fact that a majority of objects die young[67]. To optimize garbage collection, heap memory is managed in generations, which are memory pools holding objects of different ages (Shown as in Figure 11). Each generation has an associated type of garbage collection that can be configured to make different time, space and application pause tradeoffs.

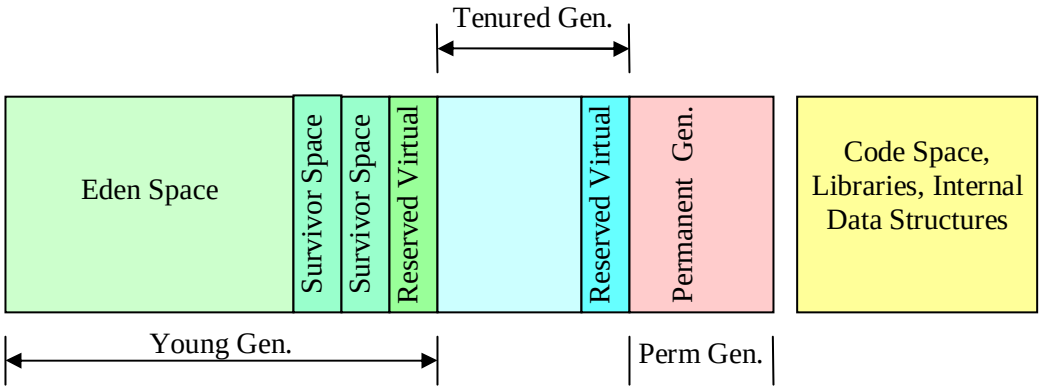


Figure 11: The default memory layout of HotSpot VM

Garbage collection happens in each generation when the generation fills up. Objects are initially allocated in the young generation. Because of infant mortality, most objects die in the young generation. When the young generation fills up it causes a minor collection. Minor collections are optimized so a young generation full of dead objects is collected very quickly. After surviving several rounds of minor garbage collections, surviving live objects are migrated to the tenured space. When the tenured generation needs to be collected, there is a major collection, which is often much slower because it involves all live objects on the heap.

In the Java HotSpot VM version 1.4.2 there are three additional garbage collectors besides the default garbage collector. Each is a generational collector that has been optimized for either the throughput of the application or low garbage collection pause times. The three collectors are named throughput, concurrent and incremental collectors. The throughput collector uses multiple threads to execute a minor collection and so reduces the serial execution time of the application. A typical use of the throughput collector is in an application that has a large number of threads allocating objects. In such an application, it is often the case that a large young generation is needed for fast allocation of many objects.

The throughput collector uses copying (sometimes called scavenge) for the minor collections such that it efficiently moves objects between two or more generations using multiple garbage collection threads. The source generations are left empty, allowing the remaining dead objects to be reclaimed quickly. For major collections, however, it uses a mark-compact garbage collection algorithm to allow generations to

be collected in place without reserving extra memory. The major collections are done by a single VM thread and are significantly slower than minor collections.

5.1.2. SPECjbb2000 Benchmark

The SPECjbb2000 (Java Business Benchmark) is a benchmark for evaluating the performance of servers running typical Java business applications[64]. The SPECjbb2000 represents an order processing application for a wholesale supplier with multiple warehouses. This benchmark loosely follows the TPC-C specification for its schema, input generation, and transaction profile. SPECjbb2000 replaces database tables with Java classes and data records with Java objects. SPECjbb2000 does no disk I/O.

The SPECjbb2000 runs in a single JVM and emulates a 3-tier system. The middle tier, which includes business logic and object manipulation, dominates the other tiers of the system. Clients are replaced by driver threads with random input to represent the first tier. The third tier is represented by binary trees rather than a separate database and database storage is implemented using in-memory binary trees.

There is a one-to-one mapping between warehouses and threads, plus a few threads for SPECjbb2000 and JVM functions. A "point" represents a two-minute measurement at a given number of warehouses. A full benchmark run consists of a sequence of measurement points with an increasing number of warehouses (and thus an increasing number of threads). The benchmark prints a metric as the numerical representation of the performance of the system besides the metric for each number of warehouses. The metric used is the throughput in terms of transactions per second.

We chose to use SPECjbb2000 for our measurements to be able to isolate the impact of our optimization techniques on the memory performance of typical Java server applications. An alternative benchmark would be the SPECjAppServer[63] benchmark. However, this benchmark tests performance for a representative J2EE application and each of the components that make up the application environment, including hardware, application server software, JVM software, database software, JDBC drivers, and the system network[63]. As such, its performance depends on many components rather than the memory subsystem.

5.2. Optimizing with Dynamic Page Migration

Prior to evaluating our new object centric optimization techniques, we quantify the impact of existing optimization techniques on the memory access locality of Java server applications. Such quantification enables us to compare the effectiveness of specialized techniques with respect to a more general technique and to verify the need for such specialized techniques.

As a general locality optimization technique, we choose dynamic page migration since this technique has been studied extensively and is known to yield performance improvements for scientific applications running on cc-NUMA servers. For our experiments, we use our dynamic page migration scheme with the same parameter values we used for the experiments described in Chapter 3.

To quantify the impact of dynamic page migration on memory access locality of SPECjbb2000, we ran SPECjbb2000 for 6, 12, 18 warehouses with and without dynamic page migration. For each number of warehouses, we counted the number of non-local memory accesses and measured the percentage reduction in the number of

non-local memory accesses due to dynamic page migration. We also measured the percentage improvement in the throughput with dynamic page migration.

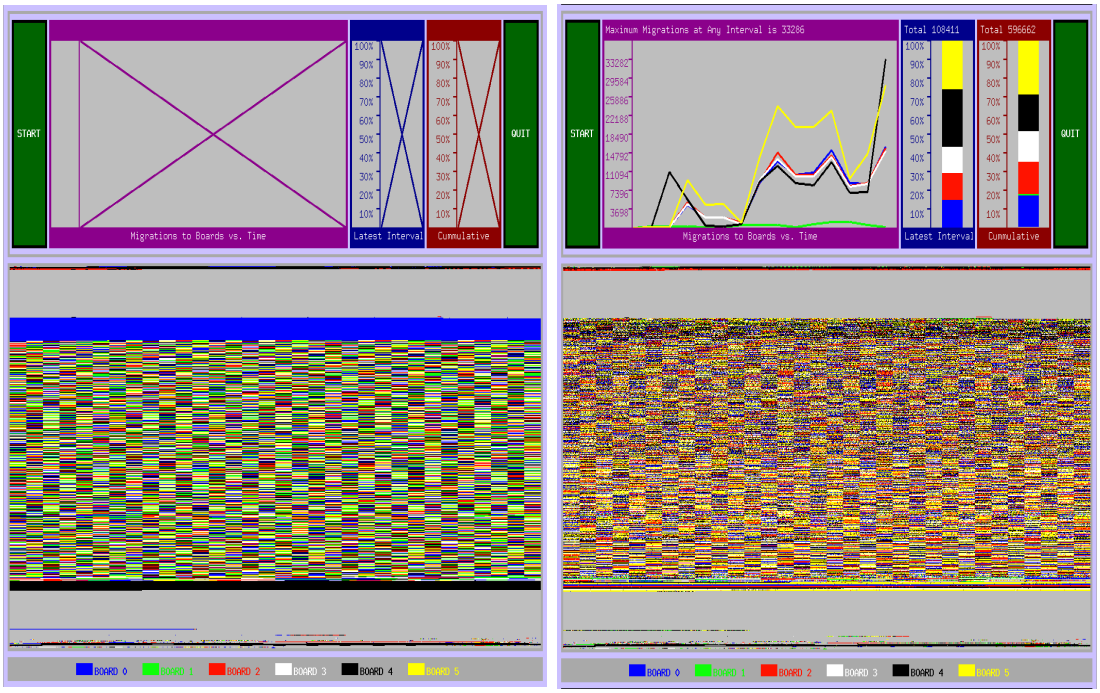
Table 10 presents the percentage reduction in the total number of non-local memory accesses when dynamic page migration is used. The second column gives the number of page migrations triggered. The third column gives the percentage of non-local memory accesses without page migration and the fourth column shows the percentages with page migration. The fifth column lists the percentage reduction in the number of non-local memory accesses when page migration is used. The sixth column gives the performance improvement in the throughput.

# of Warehouses	# of Migrations	Non-local Memory Accesses			% Improvement
		w/o Migration	with Migration	% Reduction	
6	69,796	72.0 %	52.3 %	27.4 %	-2.8 %
12	145,607	77.0 %	58.1 %	24.5 %	-3.4 %
18	165,794	77.5 %	61.3 %	20.9 %	-3.1 %

Table 10: Performance improvement due to page migration in SPECjbb2000

Table 10 shows that running SPECjbb2000 with dynamic page migration, the number of non-local memory accesses are reduced around 25% for all configurations compared to not using dynamic page migration. Moreover, it shows that the reduction percentage decreases as the number of warehouses increases due to increase in the sharing of objects among warehouses. Table 10 also shows that dynamic page migration was not able to improve the throughput despite the reduction in non-local accesses. Instead, dynamic page migration reduced throughput around 3% since the reduction in non-local accesses did not overcome the overhead introduced by migrating many pages.

To investigate the page placements with and without page migration, we also ran SPECjbb2000 when page migration GUI is enabled. Figure 12 shows the snapshots of page migration GUI presenting the locations of the virtual pages in SPECjbb200 in terms of memory units in the underlying CC-NUMA server. Figure 12(a) presents the page locations when page migration is not used, where as Figure 12(b) presents the locations of pages when page migration is used.



(a) **(b)**

Figure 12: Page placements in SPECjbb2000 without and with page migration

Figure 12(a) shows that when SPECjbb2000 runs without page migration, its pages are placed to the memory units in a finer grained fashion where small sequences of pages are placed together in memory units. This is due to the fact that HotSpot VM uses thread local allocation buffers where each thread is provided a segment of memory to allocate its objects from when space is required. Moreover, unlike scientific applications, Figure 12(b) shows that dynamic page migration was

not effective in accurately migrating pages local processors accessing those most. More importantly, Figure 12(b) shows that number of migrations triggered increases significantly throughout the execution. We suspect this is due to two reasons; First, the young generation in the Java heap is reused after every garbage collection resulting in pages changing their access patterns. Second, objects that have different access patterns are allocated in the same memory page.

Overall, Table 10 and Figure 12 show that unlike scientific applications where the reduction in the number of non-local memory accesses can be as much as 90%, dynamic page migration was not as effective in reducing the number of non-local memory accesses for SPECjbb2000. Instead, to better optimize memory access locality for this type of workload on a cc-NUMA server, we believe object level migration is needed.

5.3. Inadequacy of Page Level Optimization

Java programs tend to make extensive use of heap-allocated memory and typically have significant pointer chasing. Since typical object sizes are much smaller compared to the commonly used memory page sizes, Java applications are likely to allocate many objects in the same memory page. Moreover, if an application uniformly accesses the objects in a page, a page level memory locality optimization technique may not be as effective in reducing the number of non-local memory accesses to the page. To investigate whether page level optimization techniques, such as dynamic page migration, are too coarse grained to be effective in reducing the number of non-local memory accesses in Java server applications, it is necessary to measure the memory behavior of these applications at the object level. In this section,

we introduce a technique to measure the memory access behavior of server applications at object granularity and present result of our experiments where we measured the memory behavior of SPECjbb2000 benchmark.

5.3.1. Measuring Memory Access Locality at Object Granularity

To gather information about the object allocations by a Java application and the internal heap allocations required by the virtual machine, we modified the source code of the HotSpot VM. For each heap allocation, we inserted constructs to record the type of the allocation, the address and size of the allocation and the requestor thread. To capture the changes in the addresses during garbage collection, we also modified the source code of garbage collection modules in the HotSpot VM. For each surviving object, we record the new and the old addresses of the object.

We only instrument object allocations that survive one or more garbage collections. During each garbage collection, we map the newly surviving objects back to the corresponding original object. Since most objects die without surviving a garbage collection, we eliminate overhead due to very short lived objects.

We used the Sun Fire Link monitors to sample the address transactions during the execution of the application and later associate those transactions with the corresponding objects. Even though the information collected by the hardware monitors is sampled and incomplete, it provides sufficiently accurate profiling information. More importantly, since the monitors are implemented in hardware level, they neither interfere with the memory behavior of the application nor introduce significant overhead.

Our memory access locality measurement algorithm is a two phase algorithm. During the execution phase, we run the application on the modified virtual machine to gather information about the heap allocations and to sample the address transactions via hardware monitors. At the end of the execution phase, we generate a trace of heap allocations and memory accesses by the processors. In the post-processing phase, we process the generated trace and report measurement results.

We first instrument the executable of the virtual machine at the start of the main function to create an additional helper thread for sampling the address transactions. We use Dyninst[9] to insert the instrumentation code. Moreover, to eliminate perturbation of sampling on the address transactions, we bind the helper thread to execute on a separate processor that does not run any of the threads in the application. The helper thread initializes some instrumentation structures and samples address transactions via the Sun Fire Link monitors for the remainder of the run.

Our algorithm divides the execution of Java applications into distinct intervals. We refer to the time period from the start of a garbage collection until its termination as garbage collection interval and the time period between two consecutive garbage collections as execution interval.

We do not sample address transactions during garbage collection intervals since current virtual machines are engineered to have a small memory footprint that would likely not have a significant impact on the memory behavior of the applications. To associate address transactions with heap allocations during the post-processing phase of our algorithm, we need to store the order information for both address transactions

and allocation records. Thus, we use the index of the last sampled address transaction, which is maintained by the helper thread.

The post-processing phase combines the allocation records and address transactions recorded during each execution interval and sorts them according to the order they are requested during the execution. It then associates the address transactions with allocation records generated during the same execution interval. If a transaction is not associated with an allocation record in the execution interval being processed, the post processing phase tries to associate the same transaction with an allocation record that was recorded during an earlier execution interval.

5.3.2. Measurement Experiments

We now present the results of experiments we conducted to measure the memory access locality at the object level for SPECjbb2000 running on the HotSpot Server VM. During our experiments, we observed that SPECjbb2000 exhibits similar memory access locality regardless of the number of warehouses. Hence, we only present the results of SPECjbb2000 for 12 warehouses. In these experiments, we sampled the address transactions every 512 transactions.

Prior to describing the results of experiments, we briefly discuss the execution overhead and perturbation in SPECjbb2000 introduced by our measurements. The results of our experiments show that the throughput of SPECjbb2000 is reduced by 3% due to our source code instrumentation of HotSpot VM. In addition, we observed that 0.08% percent of all address transactions sampled are associated with the additional buffers we used to store allocation records and sampled transactions. Thus,

our measurement has neither a significant impact on the execution performance nor a significant perturbation on the memory behavior of SPECjbb2000.

Our measurement technique gathered around 10 million allocation records during the execution of SPECjbb2000 with 12 warehouses. In addition, it took around 33 million samples from the address transactions in the system interconnect. The post-processing phase of our technique associated 97.4% of the samples taken with an allocation. That is, 2.6% of all samples taken were not associated with any allocation during the execution. The majority of the unassociated address transactions fall into the code space of the HotSpot VM.

Allocation Type	# of Allocations	Memory Accesses		Non-Local Accesses	
		Count	%	Count	%
thread local buffer-TLAB	85,351	11,490,677	34.5	9,632,456	83.8
object	9	1	0.0	0	0.0
array	18	159	0.0	3	1.9
large array	1	224	0.0	0	0.0
permanent object	34,907	220,596	0.7	179,899	81.6
permanent array	9,516	15,593	0.0	11,433	73.3
scavenge survivor move	7,376,785	435,170	1.3	354,129	81.4
scavenge old move	602,940	1,849,777	5.6	1,732,677	93.7
compact move	1,932,844	14,628,184	43.9	12,107,329	82.8
active table	1	17,113	0.1	13,259	77.5
code cache	1	3,511,678	10.5	2,821,164	80.3
stack	27	125,564	0.4	102,154	81.4
memory chunks	249	35,644	0.1	18,970	53.2
jni handles	86	65,938	0.2	65,673	99.6

Table 11: Results for memory behavior of SPECjbb2000 with 12 warehouses

Table 11 presents detailed results of our experiments. In the second column, it gives the number of allocation records gathered from each type of heap allocation. The third and fourth columns give the number of memory accesses associated with

the heap allocations for the corresponding allocation type and the percentage of the associated transactions among all transactions. The fifth column gives the number of non-local accesses, and the sixth column presents the percentage of non-local memory accesses for each allocation type. Table 12 presents the results for associated transactions presented in Table 11 for each heap segment in Java heap.

Table 11 shows that the majority of allocation records we recorded were due to garbage collection of surviving objects. It also shows that there are only a few heap allocations for internal data structures used by the virtual machine itself. More importantly, Table 11 shows that accesses to heap allocated objects are mainly due to the Thread Local Allocation Buffers (TLAB) allocated from the eden space of the Java heap and the surviving objects moved into old generation during garbage collection. TLABs are the thread-local storage used by the threads for fast object allocations in the young generation.

Java Heap Section	Memory Accesses		% Non-Local
	Count	% of All Accesses	
Young Generation	11,926,231	35.8	83.7
Eden Space	11,389,586	34.2	83.8
Survivor Spaces	536,645	1.6	82.7
Old Generation	16,477,990	49.5	84.0
Permanent Generation	236,189	0.7	81.0
Internal Data Structures	3,755,937	11.3	80.4

Table 12: Memory activity per Java heap region

Table 11 and Table 12 show that around 12% of accesses are associated with the internal structures and permanent allocations by the virtual machine and 10% of these accesses are due to the code cache used for interpreter and Java method codes. That

is, even though the HotSpot VM contributes to the memory behavior of the SPECjbb2000, its contribution is not significant.

Overall, these results show that Java server applications are good candidates for memory locality optimizations due to the high percentage of non-local memory accesses. They also show that the memory behavior of SPECjbb2000 is mostly defined by the heap allocations and memory accesses it requested and the memory behavior of the Java virtual machine is not significant. Thus, locality optimization techniques that focus on optimizing the memory behavior of an application rather than the memory behavior of the underlying virtual machine hold the greatest opportunity.

5.4. Estimating Potential Benefits of Object Centric Techniques

To investigate the potential benefits of finer grain optimization techniques, in this section we present an estimation study that roughly predicts the benefits of using such optimization techniques. The estimation study is based on the heap allocations and accesses gathered during our measurement experiments.

In this study, we consider three object level placement techniques. First, *static-optimal* placement has information about all accesses to each heap allocation by processors during the execution and places objects in the memory pages local to the processors that access them most at allocation time. Second, *prior-knowledge* placement has information about the accesses to each surviving allocation during the next execution interval and moves allocations to the memory pages local to the processors accessing them most in garbage collection intervals. Third, *object-migration* placement uses object access frequencies by processors since the start of

execution up to the current time. At garbage collection, it migrates heap allocations to memory local to the processors that access those most.

In this estimation study, we measured the potential reduction in the number of non-local memory accesses for each placement technique using heap allocation records and memory accesses we gathered using our measurement tool. Figure 13 presents the percentage of non-local memory accesses in the original execution of SPECjbb2000 as well as using each placement technique.

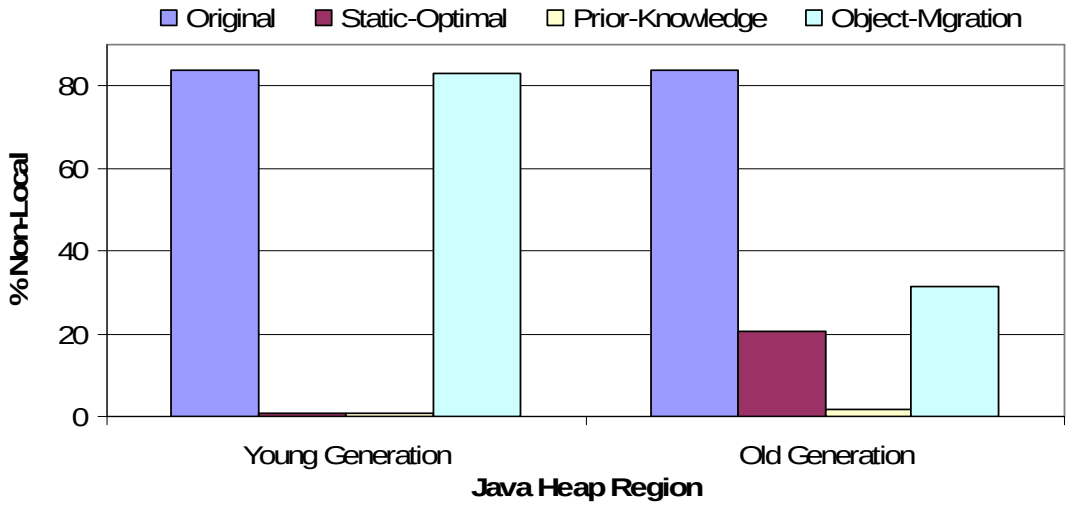


Figure 13: Potential reduction in non-local accesses for object centric techniques

Figure 13 shows that heap allocations in the young generation would significantly benefit from both static-optimal and prior-knowledge placement. It also shows that object-migration would not be effective in reducing the number of non-local memory accesses in young generation. Figure 13 indicates that the heap allocations in the old generation would also benefit from static-optimal and prior-knowledge placements. Unlike heap allocations in the young generation, allocations in the old generation however would benefit from object migrations.

Figure 13 shows that the prior-knowledge placement is more effective in the old generation compared to other placement techniques. It also shows that the static-optimal placement alone yields a significant reduction in non-local accesses in the old generation. This indicates SPECjbb2000 has some dynamically changing memory behavior in the old generation. More importantly, Figure 13 shows that dynamic object migration responds to this changing behavior quite well and yields a significant reduction in the number of non-local memory accesses in the old generation.

In Figure 13, the significant reduction in the number of non-local memory accesses in the young generation for the static-optimal placement indicates that heap allocations in the young generation are mostly accessed by single processors. Thus, we further investigated heap allocations in the young generation. We found that 94% of all accesses to the heap allocations in the young generation are requested by the same processor that requested the allocation. This is due to the fact that threads allocate their TLABs from young generation where objects are allocated from these TLABs. Moreover, since the mortality rate for the objects in TLABs are high, most of the accesses to those allocations are most likely to be from the same thread. Thus, if thread local buffers were placed local to the processor thread is running on, a substantial access locality would be possible.

More importantly, Figure 13 shows that SPECjbb2000 exhibits different memory behaviors in the young and old generations. Thus, for an object centric optimization technique to be effective, we believe it should target optimizing each generation separately and apply different techniques on each generation. To investigate the potential benefits of such a technique, we calculated the potential reduction in the

number of non-local memory accesses for a hybrid optimization technique using the fact that 94% of all observed accesses to the heap allocations in the young generation are requested by the same processors that allocated them. The hybrid optimization technique places heap allocations local to the processors that requested them in the young generation and uses dynamic object migration in old generation. We have found that such hybrid technique would reduce the number of non-local memory accesses by 73%.

5.5. Experiences with a Simple Page-Level Optimization Technique

As a result of our measurement experiments, we designed a simple optimization technique that tries to reduce the number of non-local memory accesses in the young generation using the *madvise* system call in Solaris 9. We used the move-on-next-touch feature of the madvise system call on the pages of each allocated TLAB to move the pages local to the processor that touches them next. Since each thread allocates objects from its own TLAB, the thread that allocates the TLAB is the same thread that touches those pages next.

We ran a set of experiments in which we ran SPECjbb2000 for 12 warehouses with our simple optimization enabled. In these experiments we have also recorded the number of advised pages in addition to the OS reported migration counts. Our experiments showed this simple technique was not able to improve the performance of SPECjbb2000. Instead, it reduced the throughput of SPECjbb2000 by 10.8%. Our experiments also showed that 2.4M 8K-pages are marked by the OS for migration, of which 1.7M pages are actually migrated. We observed that the reduction in the throughput of SPECjbb2000 is due to overhead introduced by the significant number

of page migrations triggered. The fact that young generation is re-used repeatedly after each garbage collection and pages in the young generation may belong to TLABs allocated by different threads at different times during the execution contributes to the number of pages migrated.

The overhead for migrating a page also includes the copying overhead of the page to its new location. However, for our simple technique, copying the page was not necessary since the pages on which we called madvise in the young generation do not contain valid data after the garbage collection iteration. Thus if the OS provided advise calls where a page would be advised but not copied, the optimization technique would benefit from such advise calls. That is, the possible advises that would only evict the page entry in the TLB would suit the needs of this optimization technique, as later when the page is touched again, OS would allocate new physical page using the first-touch policy.

5.6. Summary

In this chapter, we first evaluated the potential of existing well-known locality optimization techniques on Java server applications where we applied dynamic page migration to SPECjbb2000 benchmark. Our experiments showed that dynamic page migration was able to reduce the number of non-local memory accesses in SPECjbb2000 by only around 25% and reduced the throughput by around 3% due to the overhead introduced by migrating many pages. Unlike scientific applications, dynamic page migration was not as effective in optimizing the memory access locality in SPECjbb2000.

We also introduced an approach to measure the memory behavior of Java server applications at the object level and presented the results of our experiments where we measured the memory behavior of SPECjbb200. Overall, our results showed that Java server applications are good candidates for memory locality optimizations due to the high percentage of non-local memory accesses. Moreover, we demonstrated that optimization techniques that focus on optimizing the memory behavior of an application rather than the memory behavior of the underlying virtual machine hold the greatest opportunity.

More importantly, our experiments showed that SPECjbb2000 exhibits different memory behavior in the young and old generations. Thus, for an object centric optimization technique to be effective, we believe it should target optimizing each generation separately and apply different techniques on each generation. In the next chapter, we introduce a set of object centric techniques to optimize the memory access locality of Java server applications. These techniques work at object level and use different NUMA-aware heap layouts for young and old generations.

Chapter 6: NUMA-Aware Java Heaps

In this chapter, we introduce a set of techniques to optimize the memory access locality of Java server applications running on cc-NUMA servers. These techniques exploit the capabilities of fine grained hardware performance monitors to provide data to automatic feedback directed locality optimization techniques. We propose the use of several NUMA-aware Java heap layouts for initial object allocation and the use of dynamic object migration during garbage collection to move objects local to the processors accessing them most.

We also introduce a new approach to simulate memory behavior of parallel applications running on multiprocessor servers. Our approach is based on gathering a partial trace of memory accesses from hardware performance monitors during an actual run of an application and extrapolating it to a representative full trace to drive the simulation. Our approach also uses information on heap allocations from the memory management library used in the underlying system. Our approach is particularly suited to evaluate new software systems rather than new hardware components. Even though our approach can be used to simulate the memory behavior of various types of parallel workloads, in this thesis, we focus on Java server applications and demonstrate our approach using a hybrid execution simulator to evaluate NUMA-aware heap algorithms.

6.1. NUMA-Aware Java Heap Layouts

To optimize memory access locality of Java server applications, we propose the use of two different Java heap configurations. The first one, NUMA-Eden, uses a

NUMA-aware young generation and the current old generation of the HotSpot VM. The second one, NUMA-Eden+Old, uses both a NUMA-aware young generation and a NUMA-aware old generation.

The NUMA-Eden configuration focuses on optimizing the locality of the accesses to the objects in the young generation where as the NUMA-Eden+Old configuration focuses on optimizing the locality of the accesses to the objects in both the young and old generations. The NUMA-Eden+Old is more likely to be more effective than the NUMA-Eden since it targets all memory accesses in the application. However, it requires gathering object access frequencies by processors at runtime since unlike the eden space, the old space is not partitioned to locate memory from a single processor on a given page.

6.1.1. NUMA-Aware Young Generation

To optimize the locality of memory accesses to the objects in the young generation, we propose to divide *eden* space in the young generation into segments where each locality group of processors is assigned a segment. We do not change the layout of survivor spaces due to the fact that memory accesses to the survivor spaces throughout the execution is insignificant compared to memory accesses to eden space. In addition, we divide the eden space to equal sized segments, as shown in Figure 14.

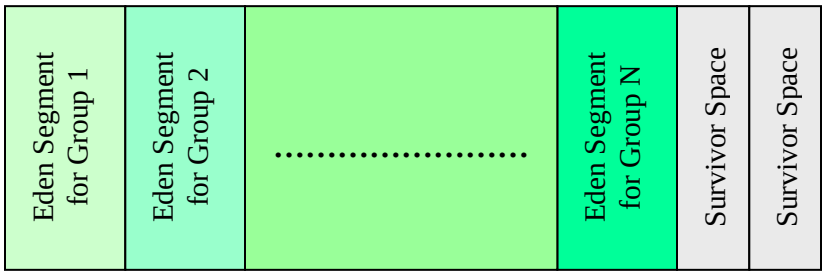


Figure 14: The NUMA-aware young generation layout

To allocate objects in the young generation in the proposed layout, the virtual machine needs to identify the processor that the requestor thread runs on, and place the object in the segment of the corresponding locality group of the processor. If application threads are bound to execute on fixed processors in the cc-NUMA server or affinity scheduling is used in the underlying OS, virtual machines can easily identify the processor an application thread runs through OS provided system calls.

When there is not enough space for object allocations in the young generation, the java virtual machine triggers minor garbage collection. For our NUMA-aware allocator, the java virtual machine may trigger minor collections in several ways; one approach is to trigger minor garbage collection when there is not enough space in a segment for object allocation. However, such an approach may trigger minor collections more often compared to original heap due to fragmentation caused by dividing the region into locality based segments. Alternatively, the virtual machine may fall back to its original behavior and allocate the objects from segments that have enough space, thus eliminating additional minor collections. Since minor collection algorithms are engineered to be fast, we believe additional minor collections will not have a significant impact on the execution performance of Java server applications. Thus, we trigger minor garbage collection when a segment does not have enough space for object allocations. Moreover, we collect all segments, even if they are not full, to eliminate future synchronization due to minor garbage collection requests by the other segments.

A NUMA-aware young generation may also have additional benefits. If garbage collection threads are bound to execute on processor groups and each collector thread

collects the dead objects in the eden segment associated with the same locality group that the thread is bound to execute, the memory access locality of garbage collection threads can be optimized. Since the garbage collection threads are known to suffer cold cache misses, such optimization can improve the performance of garbage collection threads.

6.1.2. NUMA-Aware Old Generation

Our experiments described in Section 5.3 show that almost 50% of all memory accesses are accesses to the objects in the old generation. Moreover, they also show that 84% of accesses to the objects in the old generation are non-local memory accesses. Thus, for fine grain memory locality optimization techniques to be effective for Java server applications, they should also optimize memory access locality for the objects in the old generation.

To optimize the memory access locality for the objects in the old generation, the fine grain optimization techniques should try to keep the objects local to the processors accessing them most during the lifetimes of objects. We refer to the location of an object as the preferred location if the object is placed in a memory page that is local to the processor accessing it most. To be able to identify the processors accessing the objects most, we use the transaction samples taken form hardware performance monitors as described in Section 5.3.

When an object is promoted to the old generation, it stays in the old generation during the rest of its lifetime. If the object survives another full collection after being promoted to the old generation, its address may change due to the copying collector. More importantly, the object may be accessed by different processors during the

distinct intervals of its lifetime. Thus, for a fine grain optimization technique to be effective, it should for each old generation object identify the preferred location of the object and place the object to its preferred location during garbage collection.

To optimize the locality of memory accesses to the objects in the old generation, we propose a dynamic object migration scheme. In this scheme, when an object is promoted to old generation during minor garbage collection, the preferred location of the object is identified and the object is placed in its preferred location. During full garbage collection, the preferred location of each object in the old generation is identified and the object is placed at its preferred location. Moreover, to match the dynamically changing behavior of the application, after a fixed number of minor garbage collections, the preferred locations of the objects in the old generation are re-computed and objects are migrated.

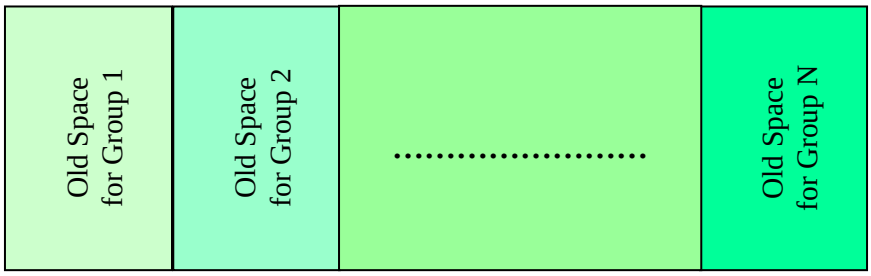


Figure 15: The NUMA-aware old generation layout

After the preferred location of an object is identified, the virtual machine needs a means to place the object in its preferred location. Thus, similar to NUMA-aware young generation, we propose to divide the old generation into segments where each locality group of processors is assigned a segment (Shown in Figure 15).

6.2. Using Hardware Monitors to Generate Parallel Workloads

To evaluate the effectiveness of the proposed heap layouts, we implemented and used a hybrid execution simulator. To drive our simulation, we generated a representative memory workload from the actual runs of the Java server application.

We chose to evaluate the proposed heap layouts using a hybrid simulator since our experiments in Chapter 4 showed that even though an instruction level simulator can reproduce memory operations executed by an application accurately, it lacks the ability to properly simulate the contention for the memory units. Moreover, in such a simulation environment it is also difficult to accurately simulate simultaneous out-of-order issue of multiple instructions by multiple processors.

Software simulation has several advantages since it enables us to reproduce the results of experiments. It also provides a means to run sensitivity experiments not possible in a live system. However, software simulation is often slow due to the need to fully simulate the underlying memory subsystem. Unlike full memory system simulators, our simulator is a hybrid simulator and executes the generated memory workload on a real system, which results in faster execution. In this section, we describe the hybrid execution simulator and the data collection and memory workload generation. This approach is possible since we need to simulate new hardware associated with the data collection, yet we are not proposing any new features of the execution of the memory system.

Our hybrid simulator has three distinct modules, as shown in Figure 16. The Workload Generation Unit generates a partial workload from an actual run of a parallel application. The Workload Scaling Unit scales the partial workload into a

larger memory workload for higher memory pressure and the Workload Execution Machine executes the workload.

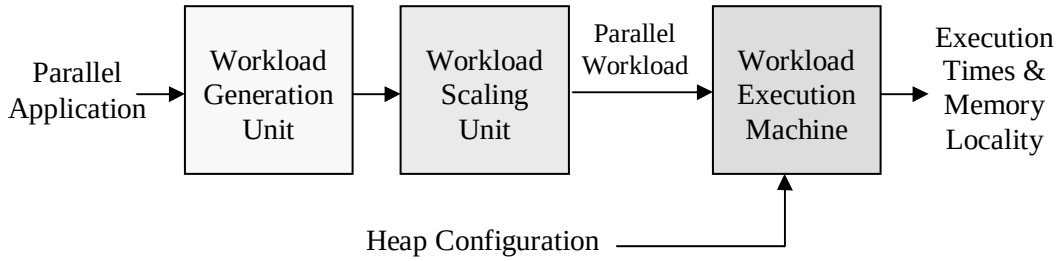


Figure 16: Flow of information in the hybrid execution simulator

6.2.1. Partial Workload Generation

Our workload generation algorithm is a two phase algorithm. During trace generation, we run an application and generate a trace of heap allocations and a set of sampled memory accesses requested during the execution. We use the memory access samples gathered from embedded hardware performance monitors and heap allocation information gathered from program instrumentation. During workload generation phase, we associate memory accesses with heap allocations in the trace and generate a partial parallel workload. For our purposes, a workload generated from an actual run of an application is a sequence of requests to allocate and access objects by processors. Accesses occur in the same order they were requested in the actual run.

We refer to an entry in a memory workload as an event. Depending on whether it is a request to allocate or access an object, we refer to an event as allocate-event or access-event, respectively. Each event in a workload must include necessary information to execute in the same way it was executed during the actual run (i.e. type and size of the object allocation or offset of the object access).

Depending on the usage of garbage collection by the parallel application being measured, garbage collection slightly complicates workload generation since some of the trace entries for heap allocations are used to track the changes in addresses of the surviving objects and they do not correspond to actual heap allocations. Thus, for these trace entries, we map the surviving objects to their source objects and use the source objects to insert access-events into the workload. That is, for each trace entry that accesses a surviving object, we insert an access-event to the workload using the allocate-event of the source object of the surviving object.

In addition, in our Java system, the HotSpot VM uses per-thread buffers for initial object allocation, which requires special handling. We insert an allocate-event into the workload for each per-thread buffer since the generated workload should allocate these buffers and the objects in them in the same order as the actual run. Moreover, for each trace entry for a surviving object, if the source object is a per-thread buffer, we insert an additional allocate-event as the source of the surviving object and use it in access-events that access the surviving object. We manipulate per-thread buffers and objects in these buffers separately to track live objects for garbage collection.

To correctly execute the workload, we need to include information on the liveness of each object at any point in the generated workload. To keep track of the liveness of objects, we also mark the last access-event by each processor to each allocate-event. Since an object can be accessed by multiple processors, we also encode the list of processors that access the object in the allocate-event of the object. Moreover, to uniquely identify objects, we assign unique identifiers to allocate-events and use them in access-events.

To accurately simulate the memory behavior of parallel applications, it is also necessary to synchronize the execution of the events by processors. If the purpose of the simulator is to evaluate hardware, synchronization at each event is required. However, when simulating runtime software systems, it is often sufficient to synchronize at natural barriers in the software. In our Java system, each garbage collection forms a natural barrier. Thus, we insert a special event for each garbage collection point to synchronize the execution of workload events by processors.

For each event inserted into the workload, we extract the information required to execute the event from the source trace entry. Each allocate-event includes information such as the size of the allocation and type of the allocation. Similarly, each access-event includes information such as the offset of the access within the allocation and the type of the access. Moreover, each event in the workload also includes the owner processor that executes the event during workload execution. However, for our Java system, we partition the events in a workload according to their locality groups rather than their processors.

6.2.2. Workload Scaling Unit

The partial workload we generate from an actual run includes all object allocations whereas it only includes a subset of all object accesses since we use sampling to gather memory accesses. Thus, the partial workload only includes a fraction of memory intensity of the actual run. To generate a workload that causes higher memory pressure, we scale the set of sampled object accesses into a larger set.

To generate a representative scaled workload from a partial workload, the scaled workload should exhibit the same memory behavior as the partial workload. To

characterize the memory behavior of the partial workload, we fit the observed distribution of object accesses to a Poisson distribution and scale the workload by a user supplied factor using this distribution. We choose Poisson distribution since the distribution of object accesses tend to have a heavy tail due to the existence of many short-lived objects and a few long-lived objects.

To fit the set of sampled object accesses to a distribution, we first classify the objects according to the number of times they are accessed in the workload. We partition objects into several buckets and histogram object accesses by access frequency. We choose the number of buckets that matches the lifetimes of objects best. Moreover, we exclude the objects whose accesses are not sampled in the distribution since we believe these objects are short lived and including accesses to these objects in the scaled workload may result in a major change in the original memory behavior.

For each histogram bucket, we record the number of objects in the class and the total number of sampled accesses to the objects. Using the ratio of the total number of accesses in each bucket to the number of all sampled object accesses, we generate a probability distribution for the set of sampled object accesses. To populate the set of object accesses, we next fit the observed distribution to a Poisson distribution, and use its density values to assign new accesses to the objects.

Since we take samples at fixed transaction boundaries, the likelihood of under sampling accesses to the objects that are accessed less frequently during the actual run is higher. To compensate for potential under-sampling error, we need to overpopulate the accesses to the objects that are accessed infrequently. By choosing a

Poisson distribution where the density values are higher for infrequently accessed objects compared to the observed distribution, we achieve the required compensation.

To insert new object accesses for a given target memory workload size, we first distribute additional object accesses among histogram buckets using the density values of the fitted Poisson distribution. Then, among all objects in each bucket, we distribute the accesses uniformly.

For each new object access, it is also necessary to assign a processor to execute the access. For some objects, we have enough samples to replicate per processor skew in data accesses, but for others we don't. Therefore, we use the total number of accesses to the object in the partial workload. If the object is accessed less than a fixed number of times, we randomly assign the processor to the new access. Otherwise, we use the ratios of the number of accesses to that object by each processor to assign the accesses. Thus we can replicate any processor affinity that may have caused that object to not have a uniform processor access pattern.

6.2.3. Workload Execution Machine

We implemented a separate program, Workload Execution Machine (WEM), to run parallel workloads generated from actual runs of the applications under a given heap configuration. WEM takes both a memory workload and a heap configuration as input and runs the workload using the given heap configuration. At termination, WEM reports the total time spent to run the workload in addition to the percentage of accesses to the heap objects that are non-local.

The WEM is composed of several units for different functionality, as shown in Figure 17. The execution unit creates a thread, workload execution thread, for each

locality group in the cc-NUMA server to execute the workload events of the corresponding locality group. Thus, each execution thread binds itself to run on its locality group throughout workload execution. The execution threads allocate objects through a common heap interface but accesses them directly from the heap. When a workload execution thread can not allocate object due to lack of available space, it notifies the virtual machine thread to start garbage collection.

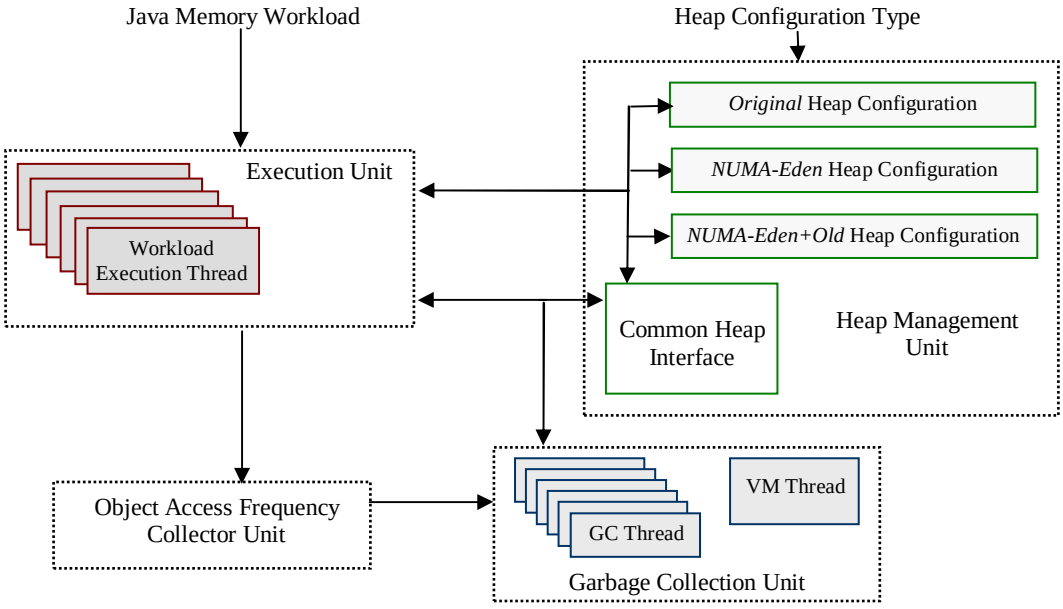


Figure 17: Workload Execution Machine architecture and the information flow

The garbage collection unit creates a GC thread for each locality group in the cc-NUMA server. Similar to workload execution threads, each GC thread binds itself to run on a locality group throughout the execution of the memory workload. The garbage collection unit also creates an additional thread for virtual machine operations, VM thread. We do not bind the VM thread to a locality group and let the OS to do the scheduling similar to the HotSpot VM. For the garbage collection unit of the WEM, we implemented the original garbage collection algorithms used by the

HotSpot VM. We implemented the parallel scavenge and mark-compact collectors for minor and major collections, respectively.

The GC threads in the garbage collection unit execute the minor garbage collection in parallel. The VM thread handles the synchronization between execution threads and GC threads. If major garbage collection is required, the VM thread executes major garbage collection itself similar to the HotSpot VM. In addition, the VM thread also runs dynamic object migration after every fixed number of minor collections. The frequency collector unit in the WEM gathers access frequencies by execution threads and propagates this information to the garbage collection unit to be used later for object migrations.

All heap configurations use a common interface to execute the heap requests by both the execution and garbage collection threads. The heap management unit takes the input heap configuration and initializes the Java heap accordingly. We implemented the heap management routines for the NUMA-aware heap configurations as well as the original heap management routines used in the Java HotSpot server VM.

6.3. Workload Generation Experiments

We ran the SPECjbb2000 with 12 warehouses and gathered a trace of heap allocations and sampled memory accesses for this run. We sampled address transaction at every 512 transactions. We gathered around 10 million allocation records and 28 million memory accesses. Using the trace gathered, we generated a partial workload in which there are around 20 million object allocations and 28 million object accesses. Among all objects allocated, there were no accesses to 42%

of them recorded in the sampled data. We believe these objects are short-lived objects and accesses to them did not get sampled. In addition, about 52.4% of objects in the workload are accessed only once whereas 5.3% objects are accessed more than once. The maximum number of accesses to an object in the sampled workload is 39,358.

Figure 18 presents the observed distribution of the object accesses in the partial workload and the Poison distribution to which we fitted the observed distribution. Figure 18 shows that we overestimate the density values for the classes that represent the short-lived objects to compensate for any potential under-sampling of accesses to these objects. Using the density measures of the fitted Poisson distribution, we scaled the initial partial workload by 16 and 32 times.

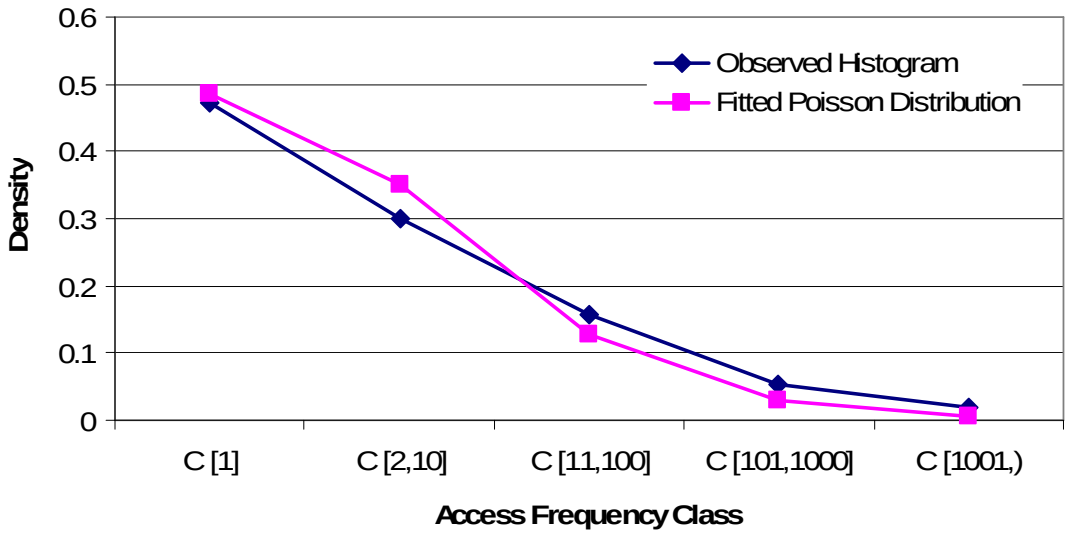


Figure 18: Observed and fitted distribution for object accesses

To investigate the impact of scaling the partial workload on memory behavior, we calculated the percentage of object accesses by locality groups in the partial and scaled workloads. Table 13 presents the percentages of the object accesses by each locality group in the scaled workloads as well as in the partial workload.

Table 13 shows that percentage of object accesses by each locality group is the same for scaled workloads independent from the scale factor. This is due to fact that we use the same fitted distribution to populate object accesses in the scaled workloads. Table 13 also shows that there are minor differences between the percentages for the partial workload and the scaled workloads. This indicates that scaling the partial workload does not change the object access behavior of the original execution significantly. We believe the difference in the percentages for the scaled workloads is due to random distribution of object accesses to the infrequently accessed objects in the partial workload.

Locality Group	Original Workload	Scaled Workloads	
		by 16	by 32
0	17.1 %	16.7 %	16.7 %
1	16.7 %	16.3 %	16.3 %
2	16.2 %	16.5 %	16.5 %
3	16.5 %	16.7 %	16.7 %
4	16.2 %	16.6 %	16.6 %
5	17.3 %	17.3 %	17.3 %

Table 13: Percentage of accesses by the locality groups

To investigate how representative the generated memory workloads are of the original execution, we measured the number of objects accesses to the young and old generations during the actual run and the execution of the partial and scaled workloads. We measured the objects accesses to the young and old generations since the ratio of accesses to these two generations is a critical parameter and one that is dependent on having accurate extrapolated traces due to its sensitivity on short-lived objects. To accurately measure the percentages for the actual run, we used the accurate counters in the Sun Fire Link hardware monitors and tracked the generation

the memory accesses went to. While these monitors can only sample individual transactions, they precisely count transactions to designated address ranges.

Table 14 presents the percentage of object accesses to young and old generations during the actual run and the execution of memory workloads using our hybrid execution simulator. Table 14 shows that the percentage of object accesses to old generations during the execution of partial workload differs by about 15% compared to the percentage during the actual run. This is due to fact that the partial workload only includes sampled accesses and accesses to short-lived objects are under sampled.

	Percentage of Object Accesses	
	Young Generation	**Old Generation**
Actual Run	51.2 %	48.8 %
Partial Workload	43.5 %	56.5 %
Workload scaled by 16	50.0 %	50.0 %
Workload scaled by 32	50.6 %	49.4 %

Table 14: Memory behavior in the young and old generations

Table 14 also shows that for the scaled workloads the percentages of object accesses to young and old generations are similar to the percentages of the actual run which also shows that our workload scaling technique is effective in generating larger workloads that exhibit similar behavior to the actual run of the application. Overall, Table 14 shows that our workload generation approach is effective in generating larger representative memory workloads from a partial workload.

6.4. NUMA-Aware Heaps Experiments

We conducted experiments using the Workload Execution Machine by running the memory workload generated from an actual run of the SPECjbb2000 for 12

warehouses on the HotSpot Server VM. To investigate the impact of higher memory pressures on the effectiveness of the proposed heap configurations, we scaled the sampled set of objects accesses by 16 and 32 times. We chose two different scaling rates to investigate the impact of memory pressure increase on the effectiveness of the proposed heap configurations.

We ran each workload generated under three heap configurations, Original, NUMA-Eden and NUMA-Eden+Old. For NUMA-Eden+Old configuration, we triggered dynamic object migration after every 3 minor collections.

The number of garbage collections triggered for the Original, NUMA-Eden, and NUMA-Eden+Old is 10, 13 and 13 respectively, of which 2 are full collections. The full garbage collections are forced garbage collections requested by the SPECjbb2000 rather than full collections triggered due to lack of heap space. The NUMA-aware heap configurations trigger more minor garbage collections compared to the original heap configuration since in these configurations, a minor collection is executed when a segment for a locality group in the young generation is full, even if the others still have available space.

6.4.1. Reduction in the Number of Non-Local Memory Accesses

We conducted a series of experiments where we ran the generated memory workloads. In these experiments we changed the underlying heap configuration and measured the percentage of the non-local memory accesses to the heap objects. Figure 19 presents the percentage of the non-local accesses to the objects allocated in young and old generations for each heap configuration compared to all accesses to the objects in each generation. It also presents the percentage of the non-local accesses to

all objects compared to all accesses. In addition, Table 15 gives the percent reduction in the number of non-local objects accesses for the NUMA-aware heaps with respect to the original heap layout.

Figure 19 shows that the percentage of non-local object accesses for the original heap configuration is over 80% for all workloads. Moreover, it also shows that our NUMA-Eden heap configuration was able to reduce the number of non-local object accesses by around 28% compared to the original heap configuration whereas the NUMA-Eden+Old configuration reduced the number of non-local object accesses by 39-41% for the workloads. Figure 19 also shows that unlike the NUMA-Eden configuration, using a NUMA-Eden+Old configuration reduces the number of non-local object accesses in the old generation. This is due to the fact that NUMA-Eden uses the original old generation where as NUMA-Eden+Old uses NUMA-aware old generation with object migration.

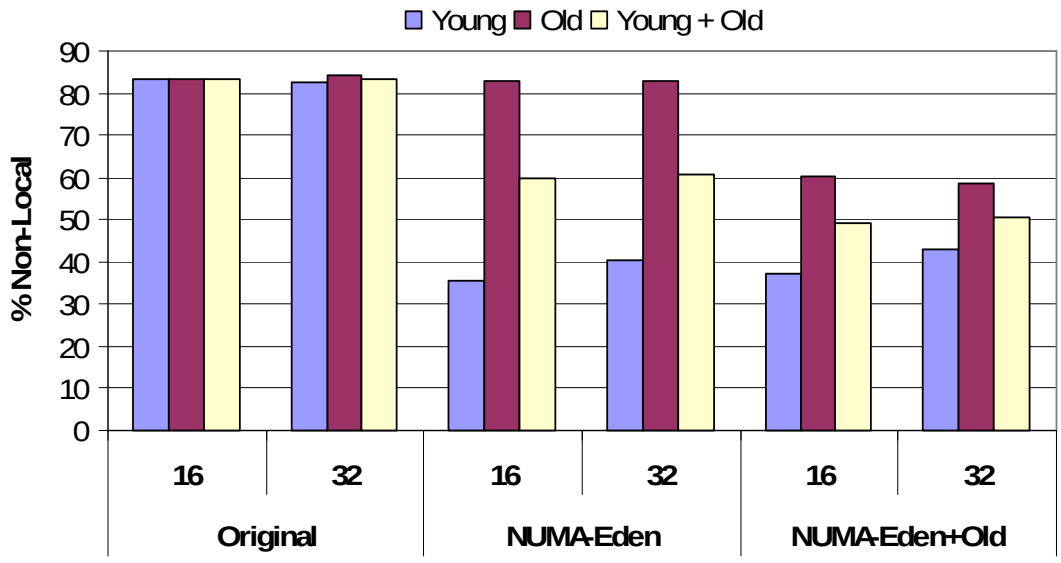


Figure 19: Non-local accesses by heap configuration for scaled workloads

Figure 19 shows that even though NUMA-Eden and NUMA-Eden+Old use the same layout for the young generation, there is a slight difference in the percentage of non-local memory accesses in the young generation for these configurations. This is due to the differences in actual page placements in the survivor spaces where we do not use a NUMA-aware layout.

Scale Factor	Heap Configuration	Young Generation	Old Generation	All Generations
16	**NUMA-Eden**	57.6 %	0.3 %	28.1 %
	NUMA-Eden+Old	55.3 %	27.5 %	41.0 %
32	**NUMA-Eden**	50.9 %	1.2 %	27.3 %
	NUMA-Eden+Old	48.0 %	30.2 %	39.5 %

Table 15: Reduction in non-local memory accesses for each heap configuration

Figure 19 and Table 15 show that NUMA-aware heap configurations are effective in reducing the total number of non-local objects accesses. They also show that using both NUMA-aware young and old generation is more effective in reducing the number of non-local object accesses for each workload compared to using only NUMA-aware young generation. Table 15 also shows that using both NUMA-aware young and old generations reduced the number of non-local object accesses in the workloads by about 40%.

In addition, Figure 19 and Table 15 show that our hybrid simulator can easily be used to evaluate the performance of the applications under different levels of memory pressure as well as different heap management algorithms. That is, our simulation approach is flexible in adjusting the memory pressure for the application by scaling the partial workloads gathered during an actual run as well as in changing the

underlying memory management library. More importantly, this type of sensitivity analysis is difficult in a real system with a fixed workload.

6.4.2. Execution Times of Memory Workloads

For each experiment, we also measured the total time spent to execute each memory workload. Note that in the memory workloads we generate, we only include object allocations and accesses requested in the original execution (as described in Section 6.2) to isolate the impact of NUMA-Aware heaps on the memory performance of the applications. Thus, the execution times of the memory workloads gathered from our simulator evaluate the potential of the NUMA-Aware heap layouts on the memory time of the applications rather than evaluating the potential of these layouts on the overall execution times of the applications.

Figure 20 presents the normalized execution time of the memory workloads for each heap configuration with respect to the execution times of the memory workloads for original heap configuration. Figure 20 also presents the normalized time spent for garbage collection. The bottom segment of each bar is just the execution time spent to run each memory workload whereas the top segment of each bar is for the time spent to execute garbage collections.

Figure 20 shows that the garbage collection times for NUMA-Eden and original heap configurations are comparable even though NUMA-Eden triggers more minor garbage collections. This is due to the fact that minor garbage collection is fairly cheap since it both is executed by multiple GC threads in parallel and does not copy many objects due to the high mortality rate of young objects.

Figure 20 also shows that for each memory workload, both NUMA-Eden and NUMA-Eden+Old configurations outperform the original heap configuration in terms of the execution time of the workload. While the NUMA-Eden reduces the execution times of the memory workloads by up to 27%, the NUMA-Eden+Old reduces the execution times of the memory workloads by up to 40%. Moreover, it also shows that using both NUMA-aware young and old generation is more effective compared to using only NUMA-aware young generation.

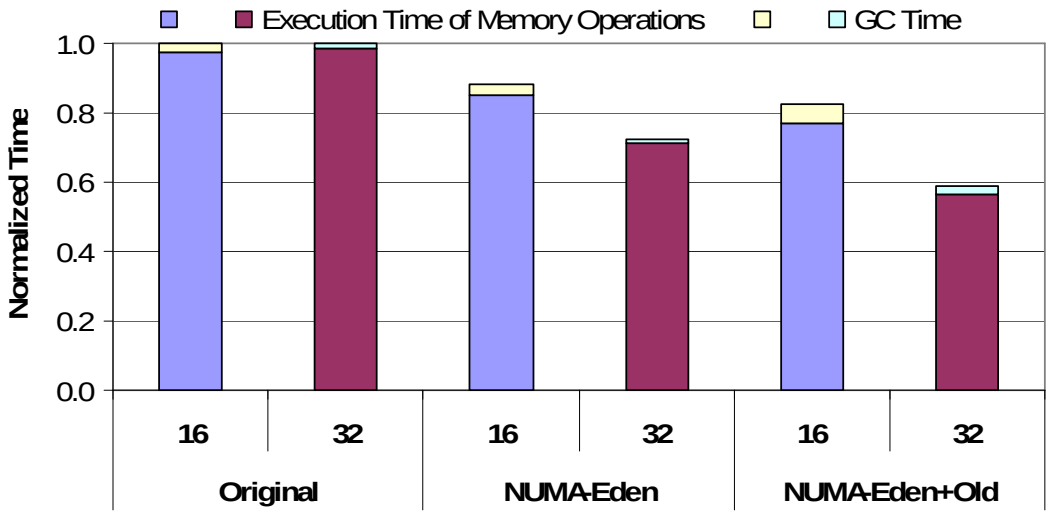


Figure 20: Normalized execution times of the memory workloads

More importantly, Figure 20 shows that as the workload size increases, the effectiveness of the NUMA-aware heap configurations increases. It shows that NUMA-aware heap configurations were able to reduce the execution time of the memory workload that is generated by scaling the original workload 16 times by around 20% compared to original heap configuration, whereas the reduction in the execution times of the memory workloads ranges by up to 40% for the workload

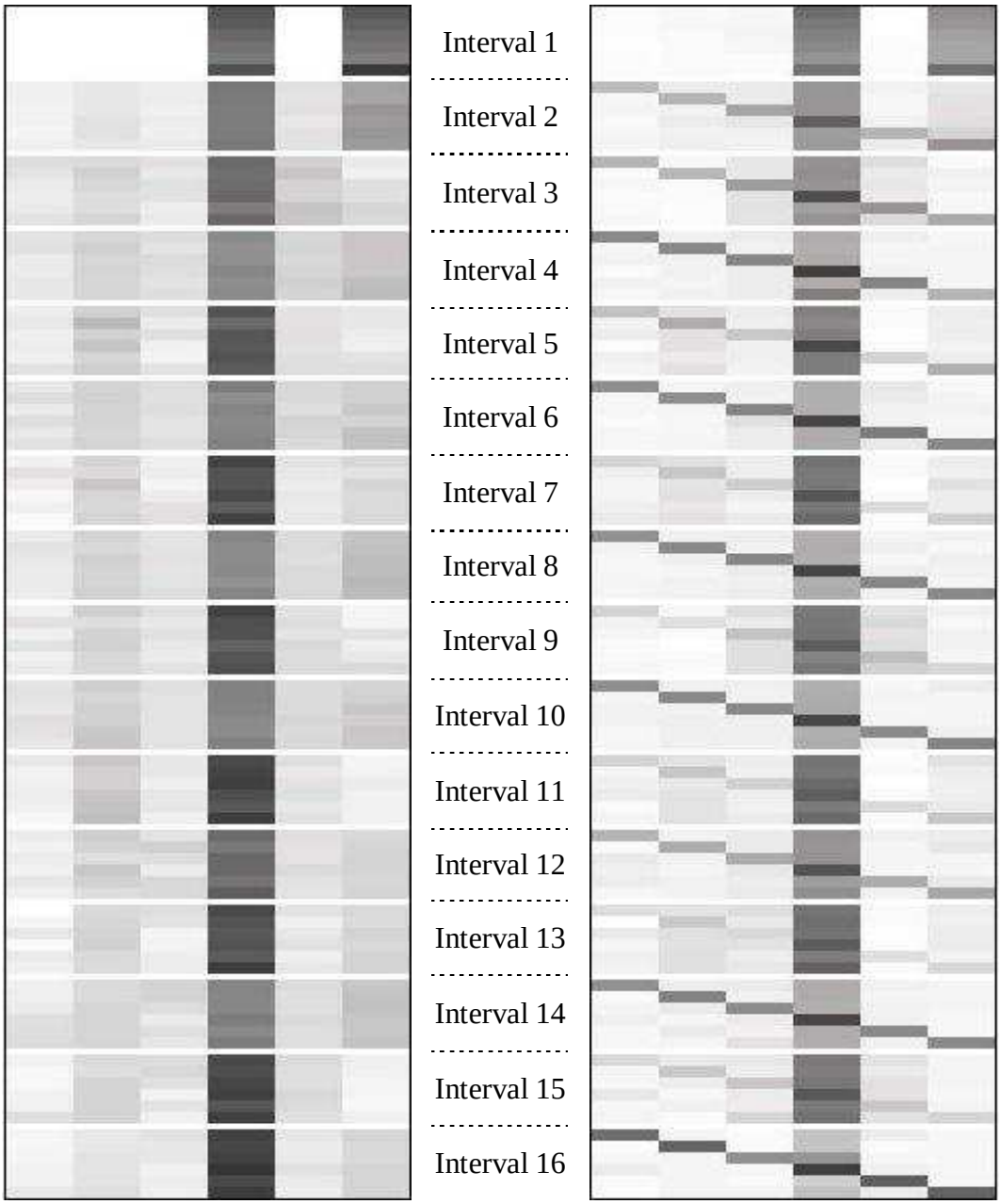
generated by a higher scaling rate of 32. Thus, Figure 20 shows that NUMA-aware heap configurations are effective in the presence of higher memory pressure.

To further investigate the impact of our NUMA-Aware heap layouts on the execution times of the memory workloads generated, we also measured the memory contention during workload execution in our hybrid simulator. To quantify the memory contention, we executed the memory workload generated under the original and NUMA-Eden+Old heap layouts and measured the percentage of memory accesses to the memory units from each locality group. Moreover, to better compare the memory contention under NUMA-aware heap layouts to original heap layout, we measured the memory contention separately for each synchronization interval in the actual run of the SPECjbb2000 where its threads synchronize.

Figure 21 illustrates the memory contention for each synchronization interval during the execution of the memory workload scaled by a factor of 32 using gray-scale color coding (the darker color represents higher memory contention). Figure 21 (a) presents the contention under the original heap layout whereas Figure 21 (b) presents the contention under our NUMA-Eden+Old layout. Moreover, for each interval in Figure 21 (a) and (b), the rows represent the locality groups where the columns represent the memory units in the system. Thus, each small rectangle cell illustrates the memory contention from a locality group to a memory unit during the interval and its color intensity indicates the significance of the memory contention.

Figure 21 (a) shows a significant memory contention on the fourth memory unit throughout the workload execution under the original heap configuration. In addition, it shows some memory contention to the second and fifth memory units even though

it is not as high as to the fourth memory unit. Figure 21 (a) also shows that memory contention on other memory units are roughly distributed uniformly. Overall, Figure 21 (a) shows that all locality groups often access the data in the same memory units, which results in higher memory contention to the memory units.



(a) Original Heap Layout **(b) NUMA-Aware Heap Layout**

Figure 21: Memory contention during the execution of the memory workload

Figure 21 (b) shows that our NUMA-aware heap layout was able to reduce some of the memory contention to the fourth memory unit as well as to the other memory units. More importantly, the diagonal contention pattern for each interval in Figure 21 (b) indicates that our NUMA-aware heap layout was able to distribute data to the memory units such that the majority of the accesses from each locality group is to the memory unit local to the group. Overall, Figure 21 (b) demonstrates that our NUMA-aware heap layout was able to reduce the memory contention to the memory units during the execution of the memory workload.

Figure 21 (a) and (b) also show that even with our NUMA-aware heap layout, there is still some contention on the fourth memory unit. We believe this is due to the long lived objects in the old generation that are uniformly accessed by all threads. We believe, these objects are placed by the virtual machine thread to the old generation during major garbage collection and are accessed uniformly by all threads throughout the execution. Therefore, these objects are not considered for dynamic object migration, which partially explains the fewer reductions in the number of non-local memory accesses in the old generation (Table 15). Moreover, Figure 21 (a) and (b)indicates that future optimization techniques should target to reduce memory contention in addition to reducing the number of non-local memory accesses in the applications.

Our experiments described in Chapter 3 demonstrated that applications have a substantial performance improvement due to elimination of memory contention to memory units in the system. Thus, Figure 21 (a) and (b) show that some of the

improvement in the execution times of the memory workloads is due to elimination of memory contention to the memory units.

To investigate the impact of our NUMA-aware heap layouts on the cache and TLB miss behavior of SPECjbb2000, we also measured the cache and TLB miss rates of the memory workload scaled by a factor of 32 under both the original and NUMA-Eden+Old heap layouts. To measure the cache and TLB miss rates, we modified our simulator to access the on-chip CPU monitors of the UltraSPARC processors in the Sun Fire system via CPU performance counter library, *libcpc*[68].

Our experiments showed that our NUMA-aware heap layout reduced the number of data TLB misses by 12.2% (from 16M to 14M TLB misses) and the number of data cache misses by around 0.2% compared to the original heap layout. We believe the substantial reduction in the number of data TLB misses is due to co-allocation of heap objects that are mostly accessed by a processor in the same memory page. Since our approach allocates and moves objects in the memory units local to the processors accessing them most, our NUMA-aware heap layouts tend to allocate objects with similar access patterns in the same memory page. We also believe the minimal reduction in the number of cache misses is due to the reduction in the number of TLB misses. Since software TLB miss handler in UltraSPARC processors relies on the external cache to read page translation entries into the Translation Storage Buffer for faster TLB miss handling[68], additional cache misses can be caused by the TLB misses. Thus, by reducing the number of TLB misses, the number of cache misses was also reduced. However, since the percentage of data cache misses due to the TLB misses is generally very small compared to all caches misses, our experiments

showed minimal reduction in the number of cache misses when our NUMA-aware heap layouts are used. More importantly, our experiments indicate that some of the improvement in the execution times of the workloads is due to reduction in the number of TLB misses under our NUMA-aware heap layout (On the average, 90 cycles is spent to handle each TLB miss in the UltraSPARC III processors, which is approximately 120ns).

Overall, our experiments show that NUMA-aware heap configurations are effective in reducing the number of non-local memory accesses and execution times of Java server workloads. Our approach was able to reduce the number of non-local memory accesses in memory workloads generated from actual runs of SpecJBB2000 by up to 41%, and also resulted in 40% reduction in the memory time of the workload. Moreover, our experiments also show that the reduction in the number of non-local memory accesses, elimination of the memory contention and the reduction in the number of data TLB misses contribute to the overall improvement in the execution times of the memory workloads under our NUMA-aware heap layouts.

More importantly, previous research[38] has shown that 25% of overall execution time is memory stall time in the SpecJBB2000. Similarly, we measured the execution time of the memory workload as around 59 seconds whereas the overall execution time of SPECjbb2000 as 191 seconds, which indicates around 31% memory time in SPECjbb2000. Thus, we expect around 10% improvement in the overall execution time of SpecJBB2000 for our approach due to around 40% improvement in the memory time of the workload.

6.5. Summary

In this chapter, we introduced new NUMA-aware Java heap layouts and dynamic object migration to optimize the memory access locality of Java server applications running on cc-NUMA servers and investigated the impact of these layouts on the memory performance of SPECjbb2000. We evaluated the effectiveness of our techniques using memory workloads generated from actual runs of SPECjbb2000.

Our proposed Java NUMA-aware heap layouts always reduced the total number of non-local object accesses in SPECjbb2000 compared to the original Java heap layout used by the HotSpot VM by up to 41%. Moreover, our proposed NUMA-aware heap layouts reduced the memory time of Java workloads derived from the SPECjbb2000 benchmark by up to 40% compared to original layout.

We have shown that using both the NUMA-aware young and old generations combined with dynamic object migration is more effective in optimizing the memory performance of SPECjbb2000 compared to using only the NUMA-aware young generation. Moreover, we have shown that as the memory pressure increases in the Java server applications, our proposed NUMA-aware heap con-figurations are more effective in improving the memory performance of Java server applications.

We also introduced a new approach to simulate memory behavior of parallel applications. Our approach gathers a partial trace of memory accesses from hardware performance monitors during an actual run of a parallel application and extrapolates it to a representative full trace. Our approach executes the generated workloads using the workload execution machine we implemented on a real multiprocessor system.

Our experiments showed that our approach generates representative workloads that exhibit similar memory behavior to the actual run.

We measured the effectiveness of several heap management algorithms under different memory pressure. Our approach has proved to be flexible in adjusting the memory pressure of the workload as well as in testing the effectiveness of different underlying system software libraries. These types of studies would be difficult in a real system with a fixed workload.

Chapter 7: Conclusions

In this thesis, we introduced several techniques to dynamically increase the locality of memory accesses in scientific and Java server applications running on cc-NUMA systems. These techniques place memory pages and heap allocated objects at their preferred memory locations identified at runtime using profiles gathered from hardware performance monitors.

To evaluate the effectiveness of the techniques described in this thesis, we used a Sun Fire 6800 server which has only small differences between local and non-local memory access times (225ns vs. 300ns) as the target machine. We believe our techniques described will be even more effective in improving the performance of the applications running on larger cc-NUMA servers such as the Sun Fire 15K[14] and SGI Altix 3000[76]. In these larger cc-NUMA servers, the data transfer times differ significantly between local and non-local memory accesses. The overall conclusions of this thesis can be summarized as follows:

Dynamic Page Migration

Our dynamic page migration approach always reduced the total number of non-local memory accesses in the applications we tested compared to their original executions. We achieved reductions up to 90%, which resulted in up to a 16% improvement in their execution times compared to their original executions. Our results demonstrated that the combinations of inexpensive plug-in monitors that sample interconnect transactions and a simple migration policy can be effectively

used to improve the performance of real scientific applications even on systems with small remote to local memory latency ratios.

The effectiveness of our page migration approach showed the advantage of putting the page migration policy at the user level while only relying on the operating system kernel to provide the actual migration mechanism. We demonstrated the importance of inexpensive hardware monitors in automatic performance tuning of the applications. We believe this type of hardware monitors combined with our page migration approach will be of increasing utility as memory systems become more complex.

Dedicated Hardware Monitors for Dynamic Page Migration

We conducted a simulation based study where we investigated the use of several different types of hardware monitors and compared their effectiveness in terms of the reduction in the number of non-local memory accesses, number of page migrations triggered and execution times. We also designed a hypothetical hardware feature to accurately gather page access frequencies specifically for dynamic page migration and compared its effectiveness to the other sources of profiles.

Our experiments showed that the reduction in the number of non-local memory accesses in the applications ranges up to 87.3% compared to their original executions, which resulted in up to an 18.3% improvement in the execution time. Overall, our experiments showed that using interconnect transactions performs similar to using cache miss information, and using ATC information performs similar to using TLB content. Using TLB miss information however performs poorly compared to other sources of profiles. Moreover, our experiments showed that the effectiveness of using

profiles other than TLB miss information was comparable to the effectiveness of perfect profiles gathered from all actual memory accesses in the executions.

More importantly, our experiments demonstrated that cache miss profiles gathered from on-chip hardware monitors, which are typically available in current micro-processors, can be effectively used to guide dynamic page migrations in an application.

NUMA-Aware Java Heap Layouts

We evaluated the potential of existing page-level locality optimization techniques on a Java server application and demonstrated that coarse-grain page-level optimization techniques are not as effective in reducing the number of non-local memory accesses in Java server applications.

Instead, we introduced new object-centric optimization techniques that use several new NUMA-aware Java heap layouts and dynamic object migration for Java server applications. We also introduced a new approach to simulate memory behavior of parallel applications using profiles gathered from hardware monitors. We evaluated the effectiveness of our techniques using the memory workloads generated from actual runs of SPECjbb2000 and running these workloads on the workload execution simulator we implemented on a real multiprocessor system.

Our NUMA-aware heap layouts always reduced the total number of non-local object accesses in SPECjbb2000 compared to the original Java heap layout used by the HotSpot VM. Reductions ranged up to 41%. Moreover, they reduced the execution times of Java memory workloads generated from actual runs of SPECjbb2000 by up to 40% compared to the original layout. Moreover, we have shown that as the memory pressure increases in the Java server applications, our

proposed NUMA-aware heap configurations are more effective in improving the memory performance of Java server applications.

We also demonstrated that hardware monitors can also be used to generate representative parallel workloads to simulate the memory behavior of parallel applications by gathering partial trace of memory accesses during an actual run and extrapolating it to a full trace. Our approach has proved to be flexible in adjusting the memory pressure of the workload as well as in testing the effectiveness of different underlying system software libraries.

Bibliography

1. Ailamaki, A., DeWitt, D.J., Hill, M.D., Wood, D.A.: DBMSs On A Modern Processor: Where Does Time Go? 25th International Conference on Very Large Databases, 1999.

2. Barroso, L.A., Gharachorloo, K., Bugnion, E.: Memory System Characterization of Commercial Workloads. 25th ACM International Symposium on Computer Architecture, Barcelona, Spain, 1998.

3. Baylor, S.J., Devarakonda, M., et al.: Java Server Benchmarks. IBM Systems Journal, 39(1): 57-81, 2000.

4. Bennett, J.K., B.Carter, J., Zwaenepoel, W.: Adaptive Software Cache Management for Distributed Shared Memory Architectures. 17th International Symposium on Computer Architecture, 1990.

5. Berger, E.D., McKinley, K.S., Blumofe, R.D., Wilson, P.R.: Hoard: A Scalable Memory Allocator for Multithreaded Applications. Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, 2000.

6. Bershad, B.N., Lee, D., Romer, T.H., Chen, J.B.: Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, 1994.

7. Blanchet, B.: Escape Analysis for Object Oriented Languages: Application to JavaTM. Conference on Object-Oriented Programming Systems, Languages, and Applications, Denver, CO, 1999.

8. Bolosky, W.J., Scott, M.L., et al.: NUMA Policies and Their Relation to Memory Architecture. Architectural Support for Programming Languages and Operating Systems, San Jose, CA, 1991.

9. Buck, B.R., Hollingsworth, J.K.: An API for Runtime Code Patching. The Journal of High Performance Computing Applications, 14, 2000.

10. Buck, B.R., Hollingsworth, J.K.: Using Hardware Performance Monitors to Isolate Memory Bottlenecks. IEEE/ACM SC'00, Dallas, TX, 2000.

11. Bull, J.M., Johnson, C.: Data Distribution, Migration and Replication on a cc-NUMA Architecture. European Workshop on OpenMP, Rome, Italy, 2002.

12. Calder, B., Krintz, C., John, S., Austin, T.: Cache-Conscious Data Placement. Architectural Support for Programming Languages and Operating Systems, San Jose, CA, 1998.

13. Chandra, R., Devine, S., et al.: Scheduling and Page Migration for Multiprocessor Compute Servers. Architectural Support for Programming Languages and Operating Systems, San Jose, CA, 1994.

14. Charlesworth, A.: The Sun Fireplane System Interconnect. ACM/IEEE SC'01, Denver, CO, 2001.

15. Chilimbi, T.M., Davidson, B., Larus, J.R.: Cache-Conscious Structure Definition. ACM SIGPLAN Programming Language Design and Implementation, Atlanta, GA, 1999.

16. Chilimbi, T.M., Hill, M.D., Larus, J.R.: Cache-Conscious Structure Layout. ACM SIGPLAN Programming Language Design and Implementation, Atlanta, GA, 1999.

17. Chilimbi, T.M., Larus, J.R.: Using Generational Garbage Collection to Implement Cache-conscious Data Placement. International Symposium on Memory Management, Vancouver, Canada, 1998.

18. Choi, J.-D., Gupta, M., et al.: Escape Analysis for Java. Conference on Object-Oriented Programming Systems, Languages, and Applications, Denver, CO, 1999.

19. Chow, K., Bhat, M., Jha, A., Cunningham, C.: Characterization of JavaTM Application Server Workloads. 4th Annual Workshop on Workload Characterization, Austin, TX, 2001.

20. Compaq Computer Corporation: Alpha Architecture Handbook (Version 4), 1998.

21. Cox, A.L., Fowler, R.J.: Adaptive Cache Coherency for Detecting Migratory Shared Data. 20th Annual International Symposium on Computer Architecture, San Diego, CA, 1993.

22. DeRose, L., Ekanadham, K., Hollingsworth, J.K., Sbaraglia, S.: SIGMA: A Simulator Infrastructure to Guide Memory Analysis. IEEE/ACM SC'02, Baltimore, MD, 2002.

23. Ding, C., Kennedy, K.: Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. Conference on Programming Language Design and Implementation, 1999.

24. Domani, T., Goldshtein, G., et al.: Thread-local Heaps for Java. International Symposium on Memory Management, Berlin, Germany, 2002.

25. ECperf Home Page: ECPerf Benchmark. http://java.sun.com/j2ee/ecperf/.

26. Edler, J.: Dinero IV:Trace-Driven Uniprocessor Cache Simulator. http://www.cs.wisc.edu/~markhill/DineroIV.

27. Eggers, S.J., Keppel, D., Koldinger, E.J., Levy, H.M.: Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. ACM SIGMETRICS, 1990.

28. Gay, D., Steensgard, B.: Fast Escape Analysis and Stack Allocation for Object-Based Programs. 9th International Conference on Compiler Construction, Berlin, Germany, 2000.

29. Gharachorloo, K., Sharma, M., Steely, S., Doren, S.V.: Architecture and Design of AlphaServer GS320. 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, 2000.

30. Glass, G., Cao, P.: Adaptive Page Replacement Based on Memory Reference Behavior. ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Seattle, WA, 1997.

31. Gosling, J., Joy, B., Steele, G.L.: The Java Language Specification. Addison-Wesley, 1996.

32. Hagersten, E., Koster, M.: WildFire: A Scalable Path for SMPs. Fifth IEEE Symposium on High-Performance Computer Architecture, 1999.

33. Heinrich, J.: MIPS R10000 Microprocessor User's Manual, Version 2.0. MIPS Technologies, Inc., 1996.

34. Horowitz, M., Martonosi, M., Mowry, T.C., Smith, M.D.: Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, 1996.

35. Hewlett Packard: HP 9000 Superdome Technical White Paper. http://www.hp.com/products1/servers/scalableservers/superdome/infolibrary/, 2005.

36. HP Corporation: Perfmon Project. http://www.hpl.hp.com/research/linux/perfmon, 2003.

37. Intel Corporation: Intel Itanium 2 Processor Reference Manual for Software Development and Optimization, 2002.

38. Karlsson, M., Moore, K., Hagersten, E., Wood, D.: Memory Characterization of the ECperf Benchmark. 2nd Annual Workshop on Memory Performance Issues, Anchorage, AL, 2002.

39. Karlsson, M., Moore, K.E., Hagersten, E., Wood, D.A.: Memory System Behavior of Java-Based Middleware. HPCA, Anaheim, CA, 2003.

40. Kim, J.-S., Hsu, Y.: Memory Subsystem Behavior of Java Programs: Methodology and Analysis. ACM SIGMETRICS, Santa Clara, CA., 2000.

41. Kistler, T., Franz, M.: Automated Data Member Layout of Heap Objects to Improve Memory Hierarchy Performance. ACM Transactions on Programming Languages and Systems, 22(3): 490-505, 2000.

42. Kurc, T., Uysal, M., et al.: Efficient Performance Prediction for Large-Scale Data-Intensive Applications. . International Journal of High Performance Computing Applications, 14(3): 216-227, 2000.

43. LaRowe, R.P., Ellis, C.S., Kaplan, L.S.: The Robustness of NUMA Memory Management. ACM SOSP, Pacific Grove, CA, 1991.

44. Larus, J.R.: Efficient Program Tracing. IEEE Computer, 26(5): 52-61, 1993.

45. Luo, Y., John, L.: Workload Characterization of Multithreaded Java Servers. IEEE International Symposium on Performance Analysis of Systems and Software, 2001.

46. Magnusson, P.S., Christensson, M., et al.: Simics: A Full System Simulation Platform. IEEE Computer, 35(2): 50-58, 2002.

47. Marden, M., Lu, S.-L., Lai, K., Lipasti, M.: Comparison of Memory System Behavior in Java and Non-Java Commercial Workloads. Workshop on Computer Architecture Evaluation using Commercial Workloads, Cambridge, MA, 2002.

48. IBM: IBM p5: A Highly Available Design for Business-Critical Applications. http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/p5_ras.pdf, 2005.

49. Nguyen, A.-T., Michael, M., Sharma, A., Torrellas, J.: The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures. ICCD'96, Austin, TX, 1996.

50. Noordergraaf, L., van der Pas, R.: Performance Experiences on Sun's WildFire Prototype. ACM/IEEE SC'99, Portland, OR, 1999.

51. Noordergraaf, L., Zak, R.: SMP System Interconnect Instrumentation for Performance Analysis. IEEE/ACM SC'02, Baltimore, MD, 2002.

52. Omni OpenMP Compiler Project: NAS Parallel Benchmarks OpenMP C Versions. http://phase.hpcc.jp/Omni/benchmarks/NPB, 2000.

53. Pai, V.S., Ranganathan, P., Adve, S.V.: RSIM Reference Manual version 1.0, Technical Report 9705. Dept. of Elec. and Comp. Eng, Rice University, 1997.

54. Ranganathan, P., Gharachorloo, K., Adve, S.V., Barroso, L.A.: Performance of Database Workloads on Shared- Memory Systems with Out-of-Order Processors. ACM Conference on Architecture Support for Programming Languages and Operating Systems, San Jose, CA, 1998.

55. Salcianu, A., Rinard, M.: Pointer and Escape Analysis for Multithreaded Programs. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Snowbird, UT, 2001.

56. Seidl, M.L., Zorn, B.G.: Segregating Heap Objects by Reference Behavior and Lifetime. Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, 1998.

57. Shuf, Y., Gupta, M., et al.: Creating Locality of Java Applications at Allocation and Garbage Collection Times. Conference on Object-Oriented Programming, Systems, Languages, and Applications, Seattle, WA, 2002.

58. Shuf, Y., Serrano, M.J., Gupta, M., Singh, J.P.: Characterizing the Memory Behavior of Java Workloads: A Structured View and Opportunities for Optimizations. ACM SIGMETRICS, Cambridge, MA, 2001.

59. Snavely, A., Carrington, L., et al.: A framework for performance modeling and prediction. IEEE/ACM SC'02, Baltimore, MD, 2002.

60. Standard Performance Evaluation Council: SPECint2000 Benchmark. http://www.specbench.org/osg/cpu2000/, 2000.

61. Standard Performance Evaluation Council: SPECjvm98 Benchmark. http://www.specbench.org/osg/jvm98/, 2000.

62. Squillante, M.S., Lazowska, E.D.: Using Processor-cache Affinity in Shared Memory Multiprocessor scheduling. IEEE Transactions on Parallel and Distributed Systems, 4(2), 1993.

63. Standard Performance Evaluation Council: SPECjAppServer Development Page. http://www.spec.org/osg/jAppServer, 2000.

64. Standard Performance Evaluation Council: SPECjbb2000 Benchmark. http://www.spec.org/osg/jbb2000, 2000.

65. Steensgaard, B.: Thread Specific Heaps for MultiThreaded Programs. International Symposium on Memory Management, Minneapolis, MN, 2000.

66. Sun Microsystems: SPECjbb2000 Results for Sun Fire 68000 with HotSpot Server VM on Solaris/SPARC, version 1.3.1-02. http://www.spec.org/osg/jbb2000/results/res2001q4/jbb2000-20011105-00092.html, 2001.

67. Sun Microsystems: Tuning Garbage Collection with the 1.4.2 Java Virtual Machine. http://java.sun.com/docs/hotspot/gc1.4.2/, 2003.

68. Sun Microsystems: UltraSPARC III Cu User's Manual (version 1.0), 2002.

69. Truong, D.N., Bodin, F., Seznec, A.: Improving Cache Behavior of Dynamically Allocated Data Structures. International Conference on Parallel Architectures and Compilation Techniques, 1998.

70. Veenstra, J., Fowler, R.: MINT: A Front End for Efficient Simulation of Shared-Mmeory Multiprocessors. MOSCOTS'94, Durham, NC, 1994.

71. Verghese, B., Devine, S., Gupta, A., Rosenblum, M.: Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, 1996.

72. VirtuTech Corporation: Simics Programming Guide version 2.2.4. http://www.virtutech.com/products/, 2005.

73. VirtuTech Corporation: Simics User Guide for Unix version 2.2.4. http://www.virtutech.com/products/, 2005.

74. Viswanathan, D., Liang, S.: Java Virtual Machine Profiler Interface. IBM Systems Journal, 39(1), 2000.

75. Wilson, K.M., Aglietti, B.B.: Dynamic Page Placement to Improve Locality in CC-NUMA Multiprocessors for TPC-C. ACM/IEEE SC'01, Denver, CO, 2001.

76. Woodacre, M., Robb, D., Roe, D., Feind, K.: The SGI Altix 3000 Global Shared-memory Architecture. SGI White paper, Mountain View, CA, 2003.

77. Yang, Q., Srisa-an, W., Skotiniotis, T., Chang, J.M.: Java Virtual Machine Timing probes - a study of Object Life Span and GC. 21st IEEE International Performance, Computing and Communications Conference, Phoenix, AZ., 2002.

78. Yardimci, E., Kaeli, D.: Profile-guided Tuning of Heap-based Memory Access. 2nd Workshop on Memory Performance Issues, Goteberg, Sweden, 2001.

79. Zagha, M., Larson, B., Turner, S., Itzkowitz, M.: Performance Analysis Using the MIPS R10000 Performance Counters. Supercomputing, Pittsburg, PA, 1996.